2

1. DATE	3. REPORT TYPE AND DATES COVERED
August 1991	final report

4. TITLE AND SUBTITLE

Common Prototyping Language

5. FUNDING NUMBERS

C: N00039-84-C-0211
T: 17

6. AUTHOR(S)

John McCarthy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

sponsoring agency:
SPAWAR 3241C2
Space & Naval Warfare Systems Command
Washington, D.C. 20363-5100

monitoring agency:
ONR Resident Representative
Mr. Paul Biddle
Stanford Univ., 202 McCullough
Stanford, CA 94305

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release: distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

See attached report.

14. SUBJECT TERMS	15. NUMBER OF PAGES: 69
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UL	UL	UL	UL

Sponsored by

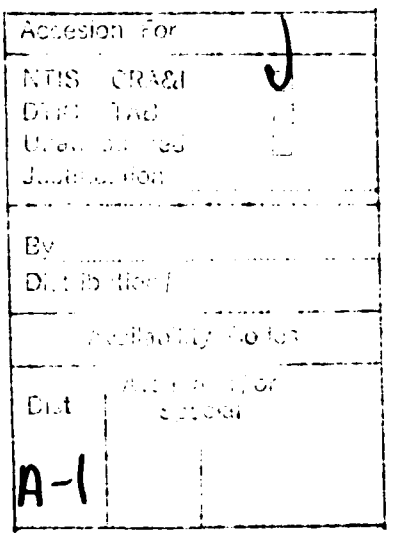
Defense Advanced Research Projects Agency (DoD)
3701 North Fairfax Drive
Arlington, VA 22203-1714

"Common Prototyping Language"

ARPA Order No. 6401

Issued by Space and Naval Warfare Systems Command

Under Contract No. N00039-84-C-0211, Task 17



Final Report on DARPA Contract Number N00039-84-C-0211 Common Prototyping Language

Richard P. Gabriel

July 23, 1991

The CPL project produced 5 results:

1. one of its members served as the technical editor for the CPL report, producing the report enclosed, which was published in Sigplan Notices
2. the group explored some language extensions that are useful for prototyping
3. the group explored an architecture for a prototyping environment that would serve as the basis for an open architecture using nearly off-the-shelf tools
4. the group explored the theoretical foundations of the key feature of the prototyping environment architecture
5. the group explored some ideas in visual prototyping based on the key features of the prototyping environment

This final report includes copies of all publically presented or published material that is a result of this project. The paper "Using CLOS-like Concepts in a Prototyping System" was presented at the Common Lisp Object System Workshop, ACM Conference on Object-oriented Programming Systems, Languages, and Applications, October 1989. The paper "Ten Ideas for Programming Language Design" was presented as the keynote address at the High Performance and Parallel Computing in Lisp Conference in Twickenham, England, November 1990. It also will be published in a forthcoming book. Material from both papers was presented at the Second Workshop on Programming Languages and Compilers for Parallel Computing in Champaign-Urbana, August, 1989.

Some of the material in "Annotations and Functors" has been presented in the above conferences. The architecture very roughly described in this paper has been transferred to a commercial group which has implemented it.

The other material has not been presented or published anywhere.

Draft Report
on
Requirements for a Common Prototyping System

Chairman: Robert Balzer
Editor: Richard P. Gabriel

Common Prototyping Working Group: Frank Belz, Robert Dewar, David Fisher, John Guttag, Paul Hudak, Mitchell Wand.

Draft Dated: Nov 11, 1988 17:12

CONTENTS

Executive Summary

Prototyping is the process of writing programs for the purpose of obtaining information prior to constructing a production version. Prototyping is used to increase the probability that the first production version will be satisfactory. It is thus a tool for reducing risk. Prototyping differs from production programming in that efficiency and completeness are often sacrificed in the interests of rapid development and ease of obtaining information.

At present there is no widely available language or system designed explicitly to support prototyping. Prototypes are built using conventional implementation languages and software development tools.

Recently, DARPA-ISTO indicated its interest in a research and development program leading to a widely applicable and well-engineered, prototyping facility, preferably within four years. Though the primary users of this facility are to come from the Ada community, the facility is to support interoperability between prototypes and multiple implementation environments, particularly Common Lisp. This draft report identifies some of the main technical requirements of such a facility. This draft report identifies some of the main technical requirements for such a facility, and is intended to provide a basis for involving broad segments of the software community in the process of refining these preliminary requirements.

None of the individual requirements seems beyond the range of current technology. The challenge lies in bringing *together disparate technologies and integrating* them into a single coherent well-engineered system. The major difficulties are as follows:

- Wide Applicability: This forces the prototyping language to have an extremely broad scope including parallel, distributed, real-time, and knowledge-based applications.
- Multi-Language Interoperability: This forces the runtime environment to couple with target environments beyond Ada, handling their data formats, invocation and argument passing conventions, and exceptions.
- Four Year Time Frame: This limits the amount of preparatory research and new system development that can be undertaken.
- Prototyping: The unique aspects of prototyping impose additional constraints and result in new technological requirements some of which currently have no known complete solution.

Given these difficulties, the report suggests adopting an iterative approach in which interim systems are built, each embodying different subsets of the identified requirements. An important aspect of this will be to adopt a flexible, open architecture.

Background

It is widely believed that the "waterfall" software lifecycle often used commercially and required by DoD has served to exacerbate the problems of software development and maintenance by delaying the discovery of incorrect or inappropriate specifications and requirements until the testing phase that follows implementation. It has been estimated that the cost of correcting such an error or bad decision increases by a factor of 10 for each phase of the waterfall lifecycle through which it passes undetected.

One of the groups that recently considered this issue was the Defense Science Board Task Force on Military Software 1987. That Task Force observed that writing a specification is like writing a program without the benefit of a computer to help check your work. Current software systems are so complex that even the most diligent thought process cannot envision them precisely and thoroughly enough that a correct and appropriate specification can be produced. Two common errors are to specify over-rich functionality and to poorly envision user interfaces. The trade-offs that need to be made during detailed design and implementation cannot be foreseen by even the best human specifier. [Brooks 1987]

The Task Force therefore proposed the use of *operational prototypes* to refine and validate specifications through trial use and feedback, and an increased role for it within the lifecycle of military software.

However, despite the apparent importance of prototyping and despite the fact that this discrepancy had been noted in previous studies [Brooks 1987] [Packard 1986], no strong prototyping tradition exists within either the commercial or the military software communities. In fact, the importance of prototyping contrasts sharply with the existing fragmented support for prototyping. No widely available languages or environments exist specifically to support prototyping. Most prototypes are written in a conventional language, usually the expected implementation language for the final product. There are few tools for extracting information from prototypes or for integrating prototypes with other prototypes and existing systems. Finally, few tools exist for directly reusing existing components (either prototype or target language) or for encapsulating them to adapt them for reuse. In short, with few exceptions prototyping relies on conventional implementation languages and software development tools. This limits both the use of and benefits from prototyping.

Recognising this state of affairs, DARPA-ISTO indicated its interest in a research and development program leading to a widely applicable, widely accessible, well-engineered, prototyping facility supporting the development, evolution, and use of prototypes for software products ultimately to be developed in the Ada *target language*.

They invited the authors of this report to work together and with the software community to:

1. Determine whether technology exists or could be developed to build such a common prototyping system within four years and, if so,
2. Identify requirements that such an integrated language and environment ought to satisfy.

The authors of this report decided that it was important to answer these questions and that it would be beneficial to involve broad segments of the software community in the process. Toward that end, the authors organized themselves into a Common Prototyping Working Group and have prepared this draft report to provide preliminary responses to these questions to solicit comments and suggestions from the research, development, and user communities.

We have set up an electronic forum for anyone interested in providing comments and suggestions on these issues or in monitoring such discussions. Send your electronic address to CPS-REQUEST@ISI.EDU to be put on the mailing list for this forum. The forum itself is CPS@VAX.DARPA.MIL; messsages for distribution should be sent directly there. In addition, we will be hosting sessions at the Software Development Environments and Principles of Programming Languages Conferences to discuss these issues.

After a period of discussion, we expect to issue a final report. In the meantime, we see great benefit from raising these issues, involving the community, and providing access to these discussions to DARPA-ISTO and others interested in pursuing prototyping.

What is Prototyping?

Prototypes are often developed in order to determine whether some functionality may be obtained in a software product, or to determine the possibility of effectively programming some task. Some people refer to this as "exploratory programming." In such situations, prototyping is part of the discovery and invention process.

Prototypes are written in order to enhance the exploration process that goes into designing the specification of a software. In the spiral model advocated by the Defense Science Board Task Force, prototyping is used to assess risks associated with proposed solutions, algorithms, expected performance, and coding strategies. In the exploratory programming model, prototyping is used as an instrument to extend one's understanding of the problem area and solution space. We call this "prototyping to learn."

Prototypes also provide a means for a group of designers, developers, customers, and users to build a consensus and a shared context for understanding and discussing the software. Often there is a tradeoff between the operational needs for particular software and the technical capabilities of the designers and developers. Prototyping is a means of assessing the best tradeoff.

Prototyping is a tool for checking designs and for obtaining information about the nature of and requirements for the final program. Prototypes are routinely used throughout the engineering disciplines, where they are called mockups, breadboards, or simulations. Given a proposed solution to a problem, prototyping is used to answer three types of question: is it in fact a solution to the posed problem; does it have acceptable performance, production cost, and reliability; and is it a solution to the right problem?

From these considerations we are able to propose the following definition of prototyping:

> ***Prototyping** is the process of constructing software for the purpose of obtaining information about the adequacy and appropriateness of the designers' conception of a software product. Prototyping is usually done as a precursor to writing a **production** system., and a prototype is distinguished from a production system by typically being more quickly developed, more readily adapted, less efficient and/or complete, and more easily instrumented and monitored. Prototyping is useful to the extent that it enables information to be gained quickly and at low cost.*

For prototyping to be most effective, it must be possible to quickly build the prototype (possibly using existing software components), to easily gather data during its use, and to rapidly modify it based on what was learned during its use.

Primary Distinguishing Characteristics of Prototyping

We separate the discussion of prototyping into a discussion of the distinguishing characteristics of prototypes, the kinds of prototypes, and their lifecycle roles. In our discussions of these issues we will often contrast ***prototypes*** and ***prototyping*** with ***systems*** and ***programming***.

Prototyping is distinguished from other software development activities by two basic characteristics:

- ***Prototypes are used to obtain information*** about the behavior and performance of a running system. To provide such information, prototypes should be capable of being fully instrumented; the result of such instrumentation is that the execution of the prototype results in the generation of data that either supplies or enables the inference of the needed information. Examples of relevant instrumentation data are: what the prototype did, the relative performance of the components, the frequency of use of components and data structures, and information about control flow.
- ***Prototyping is a quick process.*** In a well-supported prototyping environment, producing a working prototype should take significantly less time and effort than it would to produce a working deliverable program with the same functionality.

The main things that can be sacrificed to achieve these capabilities are prototype efficiency and completeness. Neither can be sacrificed completely, but each can be sacrificed to the extent that the resulting models contain the necessary behavior and structure needed to answer the posed questions while retaining sufficient performance under instrumentation to gather that information. Occasionally one or the other cannot be sacrificed at all.

Kinds of Prototypes

Because prototypes are primarily used to obtain information, it is natural to categorise them on the basis of the kinds of information they provide: ***behavioral*** and ***structural***. In general, real prototypes combine elements of both. Over their lifetime this mix may change as the questions being asked of the prototype change. The flexibility implied by this changing mix is important.

Many of the requirements concerning ease of change, the ability to couple to other languages, and the first-class nature of many things are derived from the need for this degree of flexibility.

Behavioral prototypes model *what* the system being prototyped is supposed to do. These black-box models exhibit responses to stimuli. They include both functional models and user interface mockups and may employ a variety of mechanisms—such as table lookup and human intervention- -to provide those responses. They are the most common type of prototype and are used to test whether the user's informally stated needs have been satisfied, i.e. for validation.

Structural prototypes model *how* the system being prototyped will accomplish its black-box behavior. These clear-box models exhibit aspects of the internal structure and org ization of the system being prototyped. They are used to determine feasibility, explore design alternatives, and estimate implementation and execution costs.

Prototyping Lifecycle Roles

In the waterfall model common practice is to regard prototypes as throw-aways. There are, however, many additional benefits to treating prototyping as an iterative process that begins with system definition and continues through the development and maintenance phases. As a simplified high-level model of a system, a prototype can continue to answer many questions about a system throughout the life of that system, but only if the prototype tracks the system. Therefore, this prototyping process needs tools to help designers and programmers economically coordinate prototypes and production systems.

During the lifetime of a prototype, the concerns of different parts of the prototype may repeatedly and independently move between behavior and structure. In general, prototypes will not be able to answer every question; frequently questions about performance of the production versions of the program's components must be answered by the production versions themselves. Therefore, to be most effective in this situation, the prototyping process must be able to accommodate mixtures of production and exploratory level components. This also facilitates reuse of components, adaptation of existing systems, and prototyping components embedded in another system.

There are three basic ways of coupling the prototype into the software lifecycle:

- The prototype could be the basis for writing a specification. This is the coupling currently employed in software practice where prototypes simply provide insight for subsequent but decoupled specification and development.
- The prototype is the specification itself. As a formal model any prototype is at least a partial specification of a system. Further, since it is written in a language that is executable, it can be validated by testing. However, as a specification it is likely to be over-determined.
- The prototype is the initial implementation. Frequently, a prototype has components that are quite inefficient and so may be far removed from production versions. There is, however, the possibility of moving to a ***generative basis*** in which production implementations are derived from prototypes by applying transformations to improve efficiency. Currently, such generative

processes are almost exclusively manual, but formal processes such as transformations, annotations (compiler pragmas), and special purpose application generators are beginning to be used, and have the advantage of automatically preserving functionality.

These last two options pull prototypes in opposite directions. A prototype used as an initial implementation might be more specific and operational, while a prototype used as a specification might be more abstract and declarative. Because we believe that prototypes should be specific and operational to answer behavioral and structural questions, we have eliminated the goal of using prototypes as formal specifications. Henceforth, when we use the term specification in conjunction with prototyping, it should be understood in its informal, non-technical sense.

Relationship to Programming Languages

Almost all the experience the software community has is with *programming* languages—that is, languages used to produce operational production programs. Over the years, these languages have become higher level by incorporating abstractions whose implementations are determined by the languages' compilers, thus freeing the programmer from this level of concern. This progression will surely continue.

Many programming languages have been used for prototyping. Most prototyping experience has been pragmatic and opportunistic, making use of existing technology because no alternative existed.

A good prototyping facility must include a good prototyping language in order to be able to express both behavioral and structural prototypes. But it must also support prototyping-specific capabilities (detailed later in this report) for handling incompleteness, for supporting both early and late bindings, for permitting easy modification, for allowing both imperative and declarative styles, and for tight coupling with a target language. In addition the prototyping language may not be a good programming language because it doesn't meet production requirements for performance and/or robustness.

Ideally, no language or environment boundary should be crossed in moving from a prototype to a delivered system. But rather than doing prototyping in production languages and environments as is the practice today, we envision developing, delivering, and maintaining software products in prototyping languages and environments. This means raising our languages and environments to the prototyping level, incorporating all the capabilities described in the rest of this report, while simultaneously maintaining the current production qualities. This is one of the most promising research directions and is described more fully in the Chapter "Related Research."

Relationship to Specification Languages

The research community is developing specification languages and experimenting with integrating them into the software lifecycle. This is a young but rapidly emerging field, developing, in our view, its own distinct progression of languages. These languages differ markedly from programming languages: Rather than addressing efficiency concerns, they have abandoned efficiency

entirely in an effort to describe the desired effects and properties of computations.

We are not attempting to advance the specification language progression. An important purpose of prototyping is answering questions concerning the *dynamics* of a system, relating to both its behavior and performance. To answer those questions the prototype must be executed, and its execution must be sufficiently rapid to support gathering data. Prototyping thus emphasizes executing models, not describing them, and is therefore distinct from specification efforts.

Executable specification languages were developed to meet the needs of prototyping during specification. Experience indicates that prototyping can be done in executable specification languages, but many prototyping concerns (such as the ability to create structural prototypes, handling incompleteness, supporting both early and late bindings, allowing both imperative and declarative styles, and tightly coupling with a target language) are inadequately addressed by the predominately descriptive nature of specification languages.

Feasibility

Much research remains to be done to understand the nature of prototyping and the support it requires. One eventual outcome could be that prototypes could be evolved into deliverable systems within a single language and environment that supports both prototyping and programming.

The problem of raising our application languages and environments to the prototyping level is a long term research effort. But even if there were such a language and environment suitable for both prototyping and programming, little immediate benefit would be realised unless some *existing* application language and environment were a subset of that language. That is, because of the large investments in existing application systems, significant immediate prototyping benefits only arise from systems that support the prototyping of software that ultimately will be delivered in an existing application language.

DARPA-ISTO's short-term (four year) goal of constructing a widely applicable, widely accessible, well-engineered, prototyping facility for software ultimately to be developed in Ada satisfies this criteria. The Working Group felt that it would represent a big step forward in the availability of prototyping capabilities for the Ada community and could act as a bridge between the Ada and Common Lisp communities via its interoperability between those two languages. If properly done, such an interim system could serve as a framework in which to incorporate future advances.

This report identifies requirements that such a system *should* satisfy. They are summaried at the end of this chapter and detailed in Chapters 2, 3, and 4. Our criteria were that each was an important part of such a prototyping facility and that each was individually feasible, using either existing technology or advances we could envision from a focussed research effort within a few years.

Much of the technology needed to support these requirements already exists and has been incorporated and used in some running system, and the remainder appears manageable with respect to

both effort and time scale.

However, the challenge is in bringing together all these separate technologies and integrating them into a single coherent well-engineered system rather than in satisfying any particular individual requirement. In total, this challenge is daunting because of the breadth of issues covered. The main sources of difficulty arising from the DARPA-ISTO goal are as follows:

- **Wide Applicability:** This forces the prototyping language to have an extremely broad scope including parallel, distributed, real-time, and knowledge-based applications. Many languages have addressed these application areas individually. but few except our most general purpose languages has tried to cover them all.
- **Multi-Language Interoperability:** This forces the runtime environment to couple with target environments beyond Ada, handling their data formats, invocation and argument passing conventions, and exceptions.
- **Four Year Time Frame:** This limits the amount of preparatory research and new system development that can be undertaken.
- **Prototyping:** The unique aspects of prototyping impose additional constraints and result in new technological requirements some of which currently have no known complete solution.

To reduce some of these difficulties, the Working Group adopted an open-architecture approach in which we limited the scope of many requirements or merely noted their desirability without requiring them, with the expectation that over time more advanced versions of those capabilities could be incorporated.

Thus we envision the growth of the prototyping facility to itself follow the prototyping paradigm both during the definitional phase and during later enhancements.

Nevertheless, because we did not take overall cost and effort into account in determining requirements—only their individual technical feasibility—the aggregate scope and magnitude of the engineering is quite aggressive and unlikely to be feasible using a waterfall approach attempting to embody all these requirements.

However, by adopting the iterative prototyping approach, we believe many useful interim systems could be built, embodying different subsets of the identified requirements, which satisfy the general goal of providing immediate significant benefits by prototyping software ultimately to be delivered in Ada. Choosing appropriate subsets involves both technical and policy considerations. For instance, as the understanding of prototyping improves, these requirements will evolve and be refined. We are also concerned that the aggregate scope and magnitude of the engineering needed to satisfy too many requirements in a single system may induce too conservative a design, omitting the technical advances we feel are crucial for success.

An important task not carried out by the Working Group was to carefully evaluate existing languages and environments with respect to the set of requirements for prototyping. Although it is felt that it is unlikely that any one existing language or environment meets all the stated requirements in its current form, it is clear that a number of languages and environments, including for

example Lisp, Prolog, SETL, APL, SmallTalk, ML, and others meet many of the requirements. A careful analysis in each case will be helpful both in the task of better understanding the requirements as they stand, and in determining whether one or more of these languages or environments might serve as an appropriate design base for a prototyping facility.

Objectives

We are concerned both with the language issues involved in a prototyping system and with support for the process of developing and using prototypes. We believe the following are the key objectives to be addressed by prototyping:

- Ability to provide information: The main use of prototyping is to extract information from executing a prototype or from the process of creating it. This information includes the functionality of the prototype, its performance, its structure, and the design decisions it is based on. This information must both be comprehensible and accessible. Tools and facilities for gathering, analysing, and presenting both static and dynamic information are required.
- Ability to quickly and easily produce a prototype: Prototyping will only be used if the benefit to cost ratio is favorable. An important cost factor is the ability to use existing modules and programs as components of the prototype; another is the expressiveness of the prototyping language.
- Low Inertia: Obtaining information from a prototype sometimes requires further changes to the prototype. The language and environment must facilitate such changes by minimising the effort required to make them and perform experiments.
- Mixed Behavioral and Structural Use: To facilitate the use of prototypes throughout the iterative exploratory and definitional phases, the language must support mixtures of behavioral and structural prototypes and provide mechanisms to enable this mixture to vary over time.

To this list of prototyping objectives, we add the following growth objective:

- Anticipatory: To facilitate incorporation of future improvements in prototyping technology, a prototyping facility must be engineered to be adaptable.

Summary of Requirements and Desired Features

We have determined that the focus should be on *prototyping systems*, where a prototyping system $\mathcal{PS}$ is a prototyping language $\mathcal{PL}$, along with a prototyping environment $\mathcal{PE}$.

This section summarises our key requirements and remarks on desired features. In the following

chapters, we describe the reasoning that support these requirements and remarks, and provide the complete set of such requirements and remarks on desired features.

We *require* something when we believe that leaving it out will seriously or fatally affect the success of *PS* as a prototyping system. We remark that something is a *desired feature* when we believe that it will improve *PS* as a prototyping system. We will always use the word *shall* to mean that the statement is a requirement.

A requirement statement shall have the following format:

Requirement: ***[Requirement Format]*** All requirement statements shall have this form.

A remark regarding a desired feature will have the following format:

Remark: ***[Desired Features]*** All remarks regarding desired features should be pithy and interesting.

A System

We address in this section those issues that are not wholely contained in either *PL* or *PE*.

Architecture

The success of *PS* will depend as much on the degree of engineering put into it as on the scientific advances it represents. The primary challenge for *PS* is in integrating existing technology into a coherent well-engineered system.

But *PS* is an example of a system which is far too complex to be fully envisioned in advance. Furthermore, many of the capabilities it should ultimately contain can only be partially supported initially. It must therefore support and facilitate growth.

We note that the most successful systems in terms of evolutionary growth are those that employ an *open* architecture, facilitate user additions and modifications, and provide some facility for fortifying, integrating, and incorporating user prototypes into the evolving system. An open architecture is one with visible, coherent, open-ended internal and external interfaces. Examples of such systems include X-Windows, GNU Emacs, Lisp, Smalltalk, Unix, and Tenex.

PS shall employ an open architecture and for its implementors to accompany it with a facility for fortifying, integrating, and incorporating user prototypes into the evolving *PS*.

Multi-Language Interoperability

Prototyping does not exist in a vacuum: A typical organisation engaged in prototyping has already produced a volume of software in the target language or languages. It has a number of modules and other pieces of software that will form parts of future production software. To minimise the effort to construct prototypes and to maximise their accuracy, it is desirable to reuse these components.

Such reuse should neither be restricted to source text nor to single-language systems. That is, it should be possible to separately compile parts of a large system and use those parts in the prototype. The process of putting together fragments of existing code in a prototype is called ***composition***, and this functionality should be expressible in the prototyping language $\mathcal{PL}$ and supported by the prototyping environment $\mathcal{PE}$.

This support shall enable components written in $\mathcal{PL}$ to invoke components written in the implementation languages (Call-Out) and vice versa (Call-In), and shall support all forms of invocation supported by those implementation language as well as all value and/or exception passing mechanisms.

$\mathcal{PS}$ shall provide this coupling to both Ada and Common Lisp, and it is desirable to include other implementation languages as well. Because this facility is intended only to support prototyping composition, such coupling is required only between $\mathcal{PS}$ and such implementation languages, not between the eventual Ada-based software and these languages.

The ability to compose existing code as part of the prototype addresses the need to quickly and easily produce a prototype. It provides at least one means to mix behavioral and structural components in a prototype. Further, it provides a means for evolving a prototype into its production counterpart by substituting target language modules for prototyped modules.

We believe that $\mathcal{PL}$ should allow prototype code to be written at appropriate levels of abstraction and should enable prototypers to reuse existing components, such as Ada components. It should be possible to express incomplete prototypes in $\mathcal{PL}$ and to test and instrument them in $\mathcal{PE}$.

$\mathcal{PE}$ should integrate design, coding, testing, debugging, and information extraction.

Persistent Object Management

If our hopes about the software lifecycle are fulfilled, prototypes will continue to exist beyond their use in the specification phase. In this case, all documentation, specification, and informal data that is derived or produced during the early phases of prototyping will need to continue to exist for long periods of time. In addition, the use of prototyping throughout the software lifecycle will be facilitated if there are tools that assist in maintaining the linkages between changes in the production implementation and the prototype.

In our view, persistent object support is the means of achieving these benefits in a safe and controllable manner. We feel that prototyping will be more broadly accepted if prototyping tools facilitate the long-term nature of the software lifecycle.

Persistent object management is the management of structured values or objects that persist beyond the execution of the programs that created them. In a prototyping system there are three classes of objects that need to be persistent. One is the class of objects that prototypes create and which are subsequently used by prototypes or other systems.

The second class of objects are the prototypes themselves along with the database of information maintained by $\mathcal{PE}$. Prototyping involves the rapid evolution of programs and their components using a variety of tools provided by the prototyping system. This evolution is possible only

if there is a way to preserve the prototypes outside the individual tools that create, compose, analyze, translate, execute, or otherwise manipulate them. These tools also will maintain a set of information discovered during their execution. This information should persist beyond any prototyping session.

Among this second class are objects required by *PS* for configuration and version management.

The third class of objects are the definitions of the types and abstractions used or produced by a prototype. These should be persistent ***first-class*** objects because the validity of the data instances depend on them, and they are themselves components subject to reuse and sharing. In *PL*, it should be possible to build abstractions of any kind of entity. A first-class object is one that can be passed to or returned from a procedure, stored in a data structure, or made persistent. As these definitions evolve, facilities must exist for managing their existing persistent instances.

Thus persistent object management involves a combination of mechanisms for type integrity, object identity, version management, and configuration control.

Binding Management

Binding is the process of associating values with names. In traditional programming languages, a value is bound to a name for some period during the evaluation of a program. There are several dimensions to the binding process: Binding can occur at different times, a particular binding can occur all at once or in stages, and bindings can be mutable or immutable.

From a prototyping viewpoint, the distinctions between the various binding times not only influence performance but also the ease and rapidity of implementation and modification of programs. The more requirements imposed by the language for early binding the more difficult to describe prototypes and the smaller the set of applications that can be implemented with ease.

Binding can occur all at once or in stages: Certain properties of a value can be bound at the point of type definition, others when subtypes are defined, still more when the type of a variable is declared, and the remainder when the value is specified. The ability to specify the various properties of values separately in a succession of states permits separation of issues and increases the ease and likelihood of correctness of a modification. We believe the ability to specify bindings in many stages is even more important to prototyping languages than to programming languages.

Bindings of value to names can be mutable or immutable. Immutable bindings make it easier to analyse and reason about programs, characteristics that are very important to prototyping. On the other hand, a prototype with only immutable bindings may be harder to write and modify. A prototyping language should provide both.

This flexibility of binding management has important systemic implications on the nature of *PS*. It means that functionality that has usually been statically allocated to just one portion of the system (such as the compiler, loader, or runtime system), should now be dynamically shared between them on the basis of the particular bindings chosen by the prototyper. Consider for example type safety and abstraction security. Both are required within *PS*. However, it is possible that neither could be completely handled by a *PL* compiler as they can in a compile-time type

inferencing language if types (and abstractions) are first-class objects that can be dynamically bound to names at runtime. Hence, some runtime checking might be needed to ensure type safety and abstraction security. On the other hand, we simultaneously demand that the compiler should do as much safety and security checking as possible.

A Language

The major requirements on $\mathcal{PL}$ arise from its being a prototyping language, from the need to support dual implementation languages, and from the need for it to be widely applicable.

Prototyping Language

It must be possible to construct both behavioral and structural prototypes in $\mathcal{PL}$. Therefore, $\mathcal{PL}$ must model both the functionality and the organisational structures and interfaces of the target languages. Furthermore, it must be possible to mix behavioral and structural components and have the mixture vary over time.

Since Ada *is* imperative, we believe that $\mathcal{PL}$ should be imperative but with declarative or functional constructs. The two most prominent examples of mixing the styles are ML and FX. We make no requirements regarding non-strict (lasy) versus strict semantics. However, if a non-strict semantics is chosen for any component of $\mathcal{PL}$, we require that they be reconciled with the imperative component of the language.

To retain its operationality so that prototypes can be executed *to extract information*, no component of $\mathcal{PL}$ should be non-effective (except those involving human intervention). However, $\mathcal{PL}$ should consistently incorporate the highest level, most expressive constructs that remain effective. Mathematics represents an idealised extreme in which effectiveness is not required, but many mathematical notions are effective, and they ought to be incorporated and broadly used. To the extent that expressiveness conflicts with target language integration, we felt that expressiveness should have the greater weight, i.e. the target language ought not overly influence the design of $\mathcal{PL}$.

$\mathcal{PL}$ should also allow non-determinism (an arbitrary choice over some spectrum of possibilities) to be explicitly expressed. Furthermore, it should be designed so that it will have acceptable operational performance when implemented using existing technology—a large prototype should run no worse than an order of magnitude slower than the same program in Ada.

In $\mathcal{PL}$, it should be possible to build abstractions of many kinds of entities, such as data values, expressions, commands, declarations, and L-values. These abstractions and all data values should be first class.

There will be a single abstract syntax and a single concrete syntax for $\mathcal{PL}$. The standard concrete syntax shall be used in all publications to reduce the difficulties that are associated with multiple concrete syntaxes for a single language.

Wide Applicability

To maximise the applicability of $\mathcal{PL}$ for the wide variety of military systems that use Ada, $\mathcal{PL}$ should incorporate constructs for parallel, distributed, real-time, and knowledge-based applications. While this requirement pushes the state of the art in none of these areas individually, integrating them together into a single coherent language provides one of the most difficult challenges for the designers of $\mathcal{PL}$.

Included under wide applicability is the ability to construct system tools and components, in particular portions of $\mathcal{PS}$ itself. This implies the capability in $\mathcal{PL}$ to treat types and abstractions interpretively as runtime bindings so that $\mathcal{PS}$ tools (such as debuggers and information extractors) can operate on the types and abstractions contained in user prototypes.

We suggest that major portions of $\mathcal{PS}$ be built in $\mathcal{PL}$. It is not required in order to avoid bootstrapping problems and unrealistic efficiency requirements on $\mathcal{PL}$.

An Environment

As with $\mathcal{PL}$, $\mathcal{PE}$ has both a component that it shares with programming systems, and an additional prototyping-specific component that is unique.

Programming Environment

A *programming environment* is a set of tools for helping a programmer get a program written and running. Often, the more advanced the programming environment the more open and scalable it is, and the more integrated are its tools. By *integration* we mean that the tools are interconnected in such a way that moving from tool to tool during a programming session is easy and that tools operate on common representations.

There are two distinct parts to the generic programming component of $\mathcal{PE}$. The first is the runtime system which supports the semantics of $\mathcal{PL}$. It performs memory management, ensures runtime type consistency, provides linkage among procedures and functions written in supported languages, and supports instrumentation and monitoring. We will always refer to this aspect of $\mathcal{PS}$ as the *runtime system*. The second is the development environment, which provides programming tools like editors, debuggers, and browsers, along with the user interface, the dynamic loader, the window system, and presentation tools for instrumentation and monitoring.

The prototyping environment, $\mathcal{PE}$, should encompass the most up-to-date programming environment techniques. It should be window-based, dynamic, incremental, and well integrated. Several existing systems, including the Lisp Machine, Smalltalk, APL, Cedar, and Rational Computer's Ada environment illustrate the current state of the art.

In addition to the capabilities found in these systems, $\mathcal{PE}$ should be built on a persistent object base, should support multiple versions of prototypes (both chronological and variants), and should support multiple users.

Prototyping Environment

Extracting information is one of the two activities that distinguishes prototyping from programming. This information is used to show what, how, and how well the prototype is doing. It can also be used to determine whether the prototype is functioning as intended and, if not, why not. That is, this information is used both for testing and debugging. Supporting information extraction requires mechanisms to do the following:

- building test harnesses needed for tests
- constructing scenarios to drive tests
- writing test evaluators
- reinitialising the state of an executing prototype in preparation for a test

It should be possible to instrument the prototype and its data in such a way that the instrumentation has access to the entire state of the computation including performance information.

It should be possible to satisfy the following additional higher-level instrumentation requirements:

- ***Specification:*** It should be possible to monitor changes to data satisfying a $\mathcal{PL}$ predicate.
- ***Composition:*** The user should be able to define compound events and to instrument those compound events. The presentation of instrumentation should be in those terms specified by the user.
- ***Low Inertia:*** The user should be able to quickly and easily turn instrumentation on or off and to change what is instrumented.

It should be easy for the user to run *incomplete* prototypes and to specify and change how $\mathcal{PE}$ handles unsupplied or incomplete components. A prototype is incomplete if not all procedures, functions, or types are defined or if they are partially defined. The following are examples of how $\mathcal{PE}$ might handle unsupplied or incomplete components:

- Invoke a condition handler
- Query the user
- Entering a break loop

This list is not intended to be exhaustive.

$\mathcal{PE}$ will need to enable the prototyper to modify running prototypes by altering data structure and procedure definitions.

$\mathcal{PE}$ should support the rapid development of sophisticated user interfaces employing windowing systems, pointing devices, high resolution graphical screens, and laser output devices. Because there is a trend towards designing the user interface before designing the rest of the system, user interface support is extremely important. Requirements engineering often entails providing a user

interface that can be tried out by a sample of users. Currently there are available user interface toolkits that are designed to help programmers design user interfaces by providing commonly used components along with a user interface to customise those components. We expect that $\mathcal{PE}$ will need to supply such a user interface building kit.

Research Issues

The development of $\mathcal{PS}$ raises certain research issues related to prototyping which should be investigated in a broader context and on a longer term basis.

- ***Suitability of $\mathcal{PL}$ as a production programming language***: As described above, a good prototyping language necessarily includes a good programming language. But whether a prototyping language can/should be used as a production programming language is an important open question.
- ***Mechanised Optimisation and Translation***: Alternatively, efficient production programs in the target language might be mechanically derived from the prototypes. But much work remains to make this approach practical.
- ***Relationship between Specification and Prototypes***: It is tempting to hope that given a suitable prototyping language one could avoid separate specifications by treating the prototype as the specification. It appears, however, that specifications and prototypes are complementary. Understanding the nature of these complementary roles and determining what information needs to be contained in the specification of $\mathcal{PL}$ modules to facilitate their reuse and composition is important for the continued growth of prototyping.
- ***Indexing Reusable Libraries***: Libraries are not useful without indices. No semantics-based indexing and retrieval mechanism exists. Such a facility is sorely needed to support reuse.
- ***Parallel and Distributed Systems***: Though we require a machine model that is both parallel and distributed, we realise that it is premature to insist that the first $\mathcal{PS}$ support it. This type of machine will play a more major role in the future. Particularly important will be research into new parallel and distributed programming languages.
- ***Programming Languages and Environments***: In addition to their prototyping specific aspects, $\mathcal{PL}$ and $\mathcal{PE}$ both are envisioned to have extensive generic portions that rely heavily on existing programming languages and programming environments. Continued growth of these generic portions of $\mathcal{PL}$ and $\mathcal{PE}$ rely upon continued research in these areas.

References

[**Brooks 1987**] Brooks, Frederick P., *Report of the Defense Science Board on Military Software*, September, 1987.

[**Packard 1986**] Packard, David, *National Security Planning and Budgeting*, A Report to the President by the President's Blue Ribbon Commission on Defense Management, June 1986, May, 1986.

1. A System

Introduction

The Prototyping System *PS* comprises a prototyping language *PL* and a prototyping environment *PE* which are the subjects respectively of the next two chapters. This chapter focuses on those issues which are not wholely contained in either.

Architecture

Requirement: ***[Ada Target Language]*** ***PS*** **shall be a single prototyping system that can be used for prototyping software ultimately to be developed in Ada.**

Requirement: ***[Static Analysis]*** ***PS*** **shall support extensive static analysis of prototypes.**

Large systems are difficult to manage not only because their behavior is poorly understood by programmers, but also because their structure and composition is poorly understood. It should be possible, for example, to gather information summarising the usage of abstractions, types, and modules. It should be easy to check correctness and consistency of a prototype.

Requirement: ***[Open Architecture]*** ***PS*** **shall employ an open architecture and its implementors shall accompany it with a facility for fortifying, integrating, and incorporating user prototypes into the evolving *PS*.**

The success of *PS* will depend as much on the degree of engineering put into it as on the scientific advances it represents. The primary challenge for *PS* is in integrating existing technology into a coherent well-engineered system.

But *PS* is an example of a system which is far too complex to be fully envisioned in advance. Furthermore, many of the capabilities it should ultimately contain can only be partially supported initially. It must therefore support and facilitate growth.

We note that the most successful systems in terms of evolutionary growth are those that employ an *open* architecture, facilitate user additions and modifications, and are accompanied by some facility for fortifying, integrating, and incorporating user prototypes into the evolving system. An open architecture is one with visible, coherent, open-ended internal and external interfaces. Examples of such systems include X-Windows, GNU Emacs, Lisp, SmallTalk, Unix, and Tenex.

We envision the growth of *PS* to follow the prototyping paradigm both during the definitional phase and during later enhancements.

Multi-Language Interoperability

PS is intended to be of practical use to software developers whose principal implementation language is Ada; this is the primary *target language* for *PS*. In some cases, target language software—either already developed or developed during prototyping—can provide a rich basis for effective prototyping; in fact, the knowledge to be gained by a particular prototyping exercise may be contingent upon the use of target language software. For example, critical performance analysis might be possible only when the prototype is built using components of the production program. In other situations, certain software rapidly developed using *PS* may give rise to a need for functionally equivalent code written in the target language.

These considerations illustrate the need for some special properties of *PL* and *PS*. *PS* must accommodate not only *PL* but also the target language and other implementation languages with which it interoperates in such a way that the following requirements are satisfied:

Requirement: ***[Composition]*** ***PS*** **shall enable the construction of prototypes from components written in *PL* and any of the implementation languages with which it interoperates, including at least Ada and Common Lisp.**

PS must interoperate with both Ada and Common Lisp, and it is desirable to include other implementation languages as well. Ada interoperability is required to get target language integration, and Common Lisp interoperability is required to provide prototyping access to the knowledge-based capabilities developed by the Common Lisp community and to help create a technical bridge between the Ada and Common Lisp communities.

Because this facility is only intended to support prototyping composition, such interoperability is only required between *PS* and such implementation languages, not between the eventual Ada-based software product and these languages.

Requirement: ***[Call-Out]*** ***PS*** **shall enable the invocation of components written in any of the interoperative implementation languages by components written in *PL*; all forms of invocation provided by that interoperative implementation language shall be supported.**

Requirement: ***[Call-In]*** ***PS*** **shall enable the invocation of components written in *PL* by components written in any of the interoperative implementation languages; all forms of invocation provided by *PL* shall be supported.**

Target Language Discussion

All control and data passing protocols provided by any of the interoperative implementation languages must be supported by *PS* for either Call-Out or Call-In.

For Ada, it should be possible to define an interface between a *PL* component and an Ada package that enables the *PL* component to interact with the package using whatever protocols are established in the package specification. Thus it should be possible to have the *PL* component

invoke procedures, functions, or task entries defined in such a package specification. It should be possible to have *PS* handle any exceptions that propagate from the invocation of the Ada components. It should be possible to pass to or receive from the invoked routine a value of a type defined in the routine's signature. It should be possible to refer to the value of an object declared in the package specification within the *PL* component.

Operationally, this means that it should be possible to establish a linkage mechanism that provides the context required by the target language components and enables their interaction with *PS* components.

It should be possible to define an interface between an Ada component and a *PL* component that enables the Ada component to interact with the *PL* component as if it were another Ada component, say a package. If the mechanism were to enable emulation of any Ada package then it should be possible for the Ada component to access capabilities provided by the *PL* component according to a specification of such a package. It thus should be possible for the Ada component to invoke *PL* capabilities as if they were procedures, functions, or task entries, and have them return exceptions. It should be possible for the Ada component to pass to or receive from the *PL* component a value of a type defined in an appropriate signature.

For Common Lisp, it should be possible to define an interface between a *PL* component and a Common Lisp function or generic function. It should be possible for *PS* to handle any conditions signaled during the invocation of a Common Lisp function or generic function. It should be possible for Common Lisp to handle any conditions signaled during the invocation of a *PL* component. It should be possible to pass arguments to or to receive values from the invoked Common Lisp function or generic function. These values includes functions and closures. Thus, *PS* should provide an interface that will handle coercion of types between Common Lisp and *PL*.

It should be possible for Common Lisp programs to use *PL* components as functions, generic functions, macros, method combiners, and methods.

There are some open questions regarding the relationship of the *PL* and the Common Lisp type systems. Common Lisp types are apparent at runtime, and users can extend that system. It might make sense for *PS* to support mechanisms to import Common Lisp types to avoid a potentially expensive boundary-crossing protocol.

One problem to solve is that of notification after garbage collection. Common Lisp is, and Ada may be, a garbage collected system, and many collectors transport (i.e. change the location of) objects. *PS* may need to be notified when objects are transported by Common Lisp or Ada.

These requirements establish a rich, component-based *Call-in/Call-out* capability for *PS* that enables a highly fluid use of pre-existing interoperative implementation language components in the development of new prototypes. This is essential in order to maintain the economic incentives to use *PS*. Of particular importance is the removal of a typical multiple-language intercommunication restriction to a fixed predetermined set of base types. Such a restriction cripples the utility of typical existing mixed language systems to an extent not tolerable for widespread, large-scale prototyping.

Requirement: ***[Encapsulation]*** *PS* **shall provide the means to fully encapsulate target language or** *PL* **modules.**

An *encapsulationing wrapper* is code that is invoked before and/or after the code it encapsulates. For example, a function can be traced by encapsulating the function with code that will be invoked before and after the function. The code that runs before the function will receive the same arguments as the function, and the code that runs after the function will receive the value returned by the function. The encapsulating code might store these arguments and values in a data structure as a form of instrumentation, or it might display them as the program executes. Because encapsulating wrappers are able to form the interface to the encapsulated code, wrappers can be used to change or adapt the interface of the encapsulated code to a new situation.

The encapsulating wrappers potentially mediate every interaction of a module with its external context; they arise from the specification of every possible interface between the module and the capabilities it employs or that employ it.

These wrappers establish the mapping functions needed for full interoperability, and they facilitate splicing arbitrary computation and instrumentation at any such interface. Thus this capability supports the ability to reuse pre-existing software components for rapid development of prototypes; it also uniformly supports late binding of instrumentation functions to established intermodule interfaces within a prototype.

Requirement: ***[Reusable Target Components]*** *PS* **shall support resuable components at least as well as Ada does**

Requirement: ***[Target Language Migration]*** *PS* **shall provide automated assistance for the translation of** *PL*-**specified prototypes to the target language.**

This assistance need not be in the form of a *PL*-**to-target language compiler, but it might take the form of special advisory notations in** *PL* **analogous to Ada Pragmas.**

Prototypes specified in *PL* **and developed within the** *PS* **may embody several components that by themselves are appropriate for use in a final target language program. If they are used in the program,** *PS* **will need to be used throughout program evolution and maintenance; in many cases, using** *PL* **components directly may require the inclusion of components of** *PS* **itself within the program. While the continued use of** *PS* **during maintenance may be acceptable or even desirable, embedding** *PS* **or parts of it in the program is much less likely to be acceptable. Whenever inhibitions against such inclusion arise, a need to migrate the functionality of** *PS* **components to the target language correspondingly arises.**

In some cases, the migration is conceptually easy but involves excessive trivial work. In these cases, automated assistance of the migration effort can be a substantial help and might be necessary to gain acceptance in some communities. In other cases, where the semantics of *PL* **are much more powerful than the target language, providing direct translation capabilities may only be possible by distorting the target language version beyond the ability to maintain it. In this case, there will either be pressure to inhibit the power of** *PL* **or to ignore the maintenance costs of obscure target code. Neither of these is acceptable.**

Therefore it is not sufficient to simply compile into the target language; at least some structure of the original *PL* version of the prototype should be reflected in the target language version, and that structure should be sufficiently natural as a target language idiom to permit knowledgeable programmers to be able to change it purposefully. This is especially true of structural prototypes designed to be initial implementations of a capability that must be able to evolve naturally in the target language. In such cases the prototype has a predictable future history of extensive adaptation. To the extent that *PE* provides maintenance support tools based on both the original *PL* and the target language versions, it may be possible to maintain a lesser degree of structural conventionality in the target language code.

It is not required that auxiliary documentation (such as would normally be required for production-quality software) be automatically derivable from the *PL* prototype, although advanced environment capabilities might someday provide such capabilities.

Persistent Object Management

Requirement: ***[Persistent Store]*** ***PS*** **shall support the persistent storage of arbitrary** ***PS*** **structured values or objects.**

Persistent object management is the management of structured values or objects that persist beyond the execution of the programs that created them. Coordinating and managing these values is essential for the coherence of *PS* or any other software development environment. Though such capabilities are just beginning to be tightly integrated into programming environments as replacements for file-based systems, we feel that only by committing to such a capability would the open architecture, tight integration, and evolutionary growth requirements of *PS* be realized.

Requirement: ***[Prototype Data Persistence]*** ***PS*** **shall persistently maintain data required by a prototype, or data produced by a prototype and subsequently required by that or another prototype or system.**

Sometimes, one goal of prototyping is to gather permanent data for some other use, such as the initial or permanent parameters to some program. Alternatively, a series of prototypes might be written and run on a common base of data. *PS* should facilitate the retention and later access of data communicated between prototypes and/or target-language systems.

Requirement: ***[Prototype/Configuration Persistence]*** ***PS*** **shall persistently maintain the prototype components themselves together with relevant configuration** ***PE*** **information.**

Prototyping involves the rapid evolution of programs and their components using a variety of tools provided by the prototyping system. This evolution is possible only if there is a way to preserve the prototypes outside of the individual tools that create, compose, analyse, translate, execute, or otherwise manipulate them. These tools also will maintain a set of information discovered during their execution that should persist beyond any prototyping session. A well-designed persistent object system should address crash-proofing and multi-user systems.

Among this class of persistent objects are those required by *PS* for configuration and version management. If *PS* is to support quick development of prototypes, *PS* should remove the burden of versioning and configuration from the sphere of concern of the prototyper. However, the prototyper should be provided with tools to operate on these version and configuration spaces.

Persistence Discussion

PS should persistently maintain the definitions of the types and abstractions used or produced by a prototype. These must be persistent because the validity of the data instances depends on them, and they are themselves components subject to reuse and sharing. Whether they are included in the class of persistent data values, or persistent prototype components, or both, seems rather arbitrary. What is important is that they be maintained and managed in the persistent store. As these definitions evolve, facilities should exist for managing their existing persistent instances. This issue is discussed in the Incremental Development section of Chapter 3.

Thus persistent object management involves a combination of mechanisms for type integrity, object identity, version management, and configuration control.

Traditional programming languages have not dealt directly with these issues, but instead have depended on conventions together with external mechanisms including operating systems (files) and database systems. For a variety of reasons these approaches are inadequate for an effective prototyping system. When persistence is supported only by mechanisms outside the language, there is no way to guarantee integrity of the data involved. This is a serious problem for any application that depends on complex data that must be shared among many different programs or must be combined in a variety of ways by many different users. This is exactly the case that prevails in prototyping. Prototypes are shared by tools that are used in the prototyping process. The speed of building prototypes depends on how easily they can be composed from persistent components. That in turn requires a large repertoire of shared components, efficient mechanisms for identifying and retrieving components of interest, and automated means to ensure the compatibility of the components in a composition.

As noted, there are two places to look outside of *PS* for persistent object support: operating systems and databases. Operating systems provide persistent storage for arbitrary data, but do not protect its type or identity. Operating systems may provide some rudimentary form of version control, but typically such mechanisms are one dimensional and reflect only the development history in a single chain. Database systems provide type integrity but are typically limited to a small fixed set of base types. Databases do not usually deal with the problems of version and configuration control, nor are they typically optimised for the the performance profile needed for interactive use. Neither operating systems nor databases provide special support for programs and program components.

There are several areas of concern that should be left to the designers of *PS*. One is whether the management of persistent data should be explicit or implicit. *Implicit* persistence would render persistent every data value declared persistent, while *explicit* persistence would require specific actions to save data values in persistent storage at the desired time. Another is how to maintain identity over multiple sessions, multiple computers over networks, and multiple computers not

over networks. A third is how to best handle persistent data that is both shared and mutable, especially regarding performance of any consistency algorithms used.

Binding Management

Binding is the process of associating values with names. In traditional programming languages, a value is bound to a name for some period during the evaluation of a program. There are several dimensions to the binding process. In Ada, binding can occur at compile-time, at link-time, or during execution. In Common Lisp, binding can occur at read-time, at compile-time, at load-time, or at runtime. Binding can occur all at once or in stages. Once bound the binding can be fixed or mutable.

The distinctions among the various times at which a binding takes place is very important in traditional programming languages because of its impact on performance in execution. The later the binding the greater the execution cost. Language requirements for early binding thus shift costs from execution- to compile-time. In programming languages intended for implementation of applications that are executed many times per compilation, early binding is an important requirement. In a prototyping language, however, we expect frequent change to the program and possibly fewer executions per compilation. Thus performance trade-offs between compile-time and runtime are possibly of less significance in prototyping than in production programming.

From a prototyping viewpoint, the distinctions between the various binding times not only influence performance but also the ease and rapidity of implementation and modification of programs. The more requirements imposed by the language for early binding the more difficult to describe prototypes and the smaller the set of applications that can be implemented with ease. *PS* should not impose requirements for early binding.

Binding can occur all at once or in stages. In early programming languages most bindings were simply associations between names and complete values. As our understanding of types has improved, languages have permitted more and more stages of partial binding. In strongly typed languages for example, variables are first bound to types and later to specific values. In more modern languages such as Ada, certain properties of a value can be bound at the point of its type definition, others later at a subtype definition, still more at the time of declaration of an object (i.e., constant or variable) of that subtype; all of these are prior to association with a specific value. The ability to specify the various properties of values separately in a succession of states permits separation of issues. It also increases the ease and diminishes the error-proneness in modification. The ability to specify bindings in many stages is even more important to prototyping languages than to other programming languages.

Bindings of values to names can be fixed or alterable. This distinction goes by a variety of names: constant versus variable, immutable versus mutable, or R-value versus L-value. Regardless of the nomenclature, the use of dynamically unalterable values significantly improves our ability to analyse and reason about programs, characteristics that are very important to prototyping. At

the same time, imposed restrictions to unalterable values can so diminish the applicability of a language that it may not be practical for prototyping purposes. A prototyping language should provide facilities that encourage and enable the easy use of constant bindings while providing variable bindings for situations where their need is dictated by the application.

Prototypes are used for several different purposes which lead to a spectrum of binding-time requirements that can be in conflict with one another. The times at which decisions should be bound in the development and evolution of a prototype include the following:

- Before Modeling. In behavioral prototyping for complex applications, rapid development and acceptable performance can be achieved by having application-specific primitives that are bound to efficient implementations prior to construction of the prototype. This might be accomplished through a library mechanism that allows importation of application components implemented in other languages.
- Early Binding. Structural prototypes share with implementation languages the need to exploit early binding. To be effective for structural prototyping, a prototyping system must have mechanisms for expressing early binding properties and should have processors that can exploit those specifications in ways which reflect their time, space, or other resource costs in a full implementation.
- Late Binding. The earlier a binding decision can be expressed in a language the more it can be exploited throughout the program and therefore the more difficult and more pervasive the consequences of changing that decision later. A prototyping system should not require the use of early binding mechanisms. The binding features should permit independent specification of small granularity so that changes to a particular binding will not require redesign or restructuring of major portions of the model. Ideally, any change in the design of a prototype will require modification only at a single point in the prototyping program, though this is an ideal that can only be approached.
- Incomplete Binding. Prototypes are models that are intended to answer questions about aspects of the systems they are modeling. Thus they are seldom complete. Unlike traditional programming languages, *PL* should not require programs to be complete. *PS* and its processors should support the translation, analysis, execution, and testing of prototypes that are incomplete. Users of a prototyping system should be able to extract any behavioral, structural, or expository information that can be determined from a prototype specification independent of the degree of its completeness.

In a prototype, modifiability and flexibility in the design are generally more important than performance in execution. Therefore, *PS* should emphasise ease of modification of its programs, even when that is at the expense of performance. This is in strong contrast with programming languages such as Ada.

In a prototyping system, unlike an implementation environment, there is a high expectation for frequent change. Consequently, *PS* should not require bindings to values to be monolithic specifications, but should instead allow separate independent specification of the properties of a binding (such as its type) so that they can be identified and isolated from one another during the

evolution of a prototype.

These differing requirements for binding can be accommodated in a single *PS*. That *PS* permits incomplete specifications or specifications for which there are no known automatic translation methods does not imply that complete and/or executable programs cannot also be described in the language. That *PS* has a rich set of abstraction mechanisms, abstractions, and late-binding features does not mean that it cannot provide a compatible set of implementation features with early binding characteristics. That *PS* supports built-in or user-definable libraries of application specific prototyping primitives does not preclude simultaneous support for easily composable general purpose primitives as well.

These differing requirements for binding should be accommodated in *PS*. It is inherent in the prototyping process that the models themselves are dynamic objects which constantly undergo change in both their form and use. The various prototypes of a given system should be accessible in a common form so that they can be analysed and processed together, so that inconsistencies can be recognised, and so that they can be integrated to contribute directly to the development and maintenance of the systems they model.

Requirement: ***[Mutable and Immutable Bindings]*** *PS* **shall provide mechanisms for both mutable and immutable bindings.**

The ability to specify immutable properties enhances our ability to understand, analyse and reason about programs. The ability to specify mutable properties greatly expands the class of application that can be easily prototyped.

Requirement: ***[Early and Late Bindings]*** *PS* **shall provide arbitrarily late binding of any prototyping decisions, but must simultaneously permit specification of early binding. That is,** *PS* **must allow late binding, while permitting early binding to be described and exploited, for example, by its compilers. In addition, binding decisions regarding components of** *PS* **used in a prototype shall be permitted to made early or late.**

For behavioral purposes, it is important that *PS* **provide a rich set of mechanisms that can be easily and rapidly composed to build operational models that display the functionality and/or interface characteristics of the intended system. The goal here is neither readability nor performance in the prototype, but writability; that is, the ability to rapidly and reliably build models.**

This goal can be met in two contrasting but complementary ways. *PL* **can provide a set of very general data and control primitives that can be easily composed to build arbitrarily complex behavioral prototypes. The price for this approach is a requirement for late binding and its accompanying performance costs, and a requirement for significant user training experience before facility is gained in effective use of the system.**

The alternative, which is available only when much is known about the common characteristics of the behavioral models to be built, is to provide direct access to specialised primitives that already model the component behavior of the intended prototypes. Although more special purpose, this extremely early binding of decisions common across many prototypes can provide performance and expository benefits not otherwise achievable in rapidly prototyped behavioral models.

For structural purposes, it is important that *PS* provide a capability for explicit early binding of any decision that can affect the structure, resource requirements, or performance of its models. This clear box model of a system can be described only in a *PL* that permits detailed specification of representation and implementation decisions and that has mechanisms that can act to create efficient executable implementations of the model.

2. A Language

Introduction

The three major factors influencing *PL* are the commitments made to make it be a prototyping language, to support multiple implementation languages, and to make it widely applicable.

The first forces it to be a wide spectrum language so that both behavioral and structural prototypes can be constructed, to incorporate high-level and non-deterministic constructs while avoiding any non-effectiveness, to support multiple computational models, and to support incomplete prototypes.

The multiple implementation language commitment forces *PL*'s abstraction constructors to address data sharing among a variety of programming language implementations, but especially Ada and Common Lisp.

The wide applicability commitment forces *PL* to have an extremely broad scope to encompass parallel, distributed, real-time, and knowledge-based applications. Only the most general purpose languages have attempted such a broad scope. This scope requirement is exacerbated by also requiring *PL* to be capable of constructing system tools and components, particularly portions of *PS* itself.

This chapter identifies the individual language requirements that arise from these commitments and discusses the issues involved.

Computation Model

By *computation model* we mean the underlying mechanisms and structures through which one understands and reasons about a program, in particular prototypes written in *PL*. A computation model is by and large *operational*, akin to a virtual machine, and contrasts with a *denotational model* which gives a declarative reading of a program. All programming languages have a computation model (and possibly more than one), whether explicitly and formally stated or not, and it is usually reflected faithfully in the language itself. The computation model is important in that it can greatly influence the way one thinks about programs and, more importantly, the way one conceives solutions to problems.

Unfortunately, many of the general goals of prototyping place conflicting constraints on the computation model and language. The difficulty lies primarily in the multiple roles that prototypes might play. For example, a goal such as having low inertia or modifiability seems to conflict with the desire for safety, and the goal of incompleteness may to conflict with the option of having a prototype serve as a specification. Ideally an effective *PL* will be flexible enough to satisfy all of the resulting constraints, but this may be too much to ask. For example, a language that utilises effective type inference instead of relying solely on explicit typing satisfies the desire for both low inertia and safety with respect to typing, and may thus be a suitable language feature for a *PL*, but this kind of solution may not be feasible for each of the desirable characteristics of *PL*.

We might begin by looking at the existing classes of popular computation models. In fact there are many:

- Conventional imperative models.
- Object-oriented models.
- Purely functional (lambda calculus) models.
- Pure logic (predicate calculus) models.
- Abstract data type models.
- Equational and term-rewriting models.
- "Knowledge-based" models and other artificial intelligence models.

Rather than study these individually, we focus on certain qualitative *characteristics* that each model may have. In the following paragraphs the more interesting characteristics are examined, with an emphasis on their relationships to the goals of prototyping.

Imperative versus Declarative

This is one of the most fundamental distinctions in classes of computation models and languages. An *imperative* computation model is characterised as having an *implicit state* component that is modified or *side-effected* by constructs or *commands* in the source language. As a result, such languages generally have a notion of *sequencing* of the commands to permit precise and deterministic control over the state. Most languages in existence today are imperative, including Ada.

In contrast, a *declarative* computation model is characterised as having *no* implicit state, and thus the emphasis is placed entirely on programming with *expressions* or *terms*. State-oriented computations are accomplished by carrying the state around explicitly, rather than implicitly. The resulting programs are usually easier to reason about equationally—because referential transparency is maintained—but the lack of implicit side-effects can impact modularity in that the addition of a small piece of state might involve modifications to many parts of a program.

Requirement: ***[Blended Computation Model]*** $\mathcal{PL}$ **shall allow** both imperative and declarative or functional styles of programming.

Because Ada is imperative, we believe it will be natural for $\mathcal{PL}$ to also be imperative so that structural prototypes can be constructed. However, many applications (largely but not entirely behavioral in nature) do not require the use of side-effects, and are more succinctly prototyped using a declarative or functional style. Since $\mathcal{PL}$ must support arbitrary mixtures of behavioral and structural prototypes, its computation model should blend the two and allow one to prototype both declaratively and imperatively. For example, $\mathcal{PL}$ could be an imperative language with constraints on the way side-effects are used (such as limiting their scope) with a simpler

imperative semantics than traditional imperative languages (thus facilitating program comprehension) and with a very expressive declarative sub-language (permitting predominantly declarative programming if desired). The two most prominent examples of mixing the styles are ML and FX.

Sequential versus Parallel

Requirement: *[Concurrency]* $\mathcal{PL}$ **shall allow the expression of concurrency.**

Parallel and distributed systems are becoming increasingly important, and thus the need for expressing concurrency in the prototype may arise from the inherent concurrency of the problem specification. On the other hand, the solutions to many problems often involve the conception of concurrent or at least independent activities, regardless of whether it was part of the problem specification. Thus, $\mathcal{PL}$ should allow the expression of concurrency to support either kind of activity.

The requirement for concurrency interacts strongly with the requirement for a blended computation model. In this regard we note the following:

- The sequentiality of traditional imperative programming languages (i.e. ones without parallel constructs) is at odds with the goal of expressing parallelism, although special parallel constructs, like those in Qlisp and CSP-like solutions to the problem, like those in Ada, are feasible. If such a solution is adopted for $\mathcal{PL}$, the expression of such parallelism should be as natural and effortless as possible, and should support all levels of granularity.
- A declarative language, which is non-committal with respect to sequential or parallel evaluation, may seem like a better alternative, since the parallelism is limited only by data dependencies. However, there are still pitfalls, the most significant being that just as such languages have no explicit sequentiality, they also have no explicit parallelism—in such languages parallelism is usually left *implicit*. If such a solution is adopted for $\mathcal{PL}$ we remark that it is desirable for to include explicit operational constructs to "emphasise" the parallelism or process granularity.

Strict versus Non-Strict (Lazy) Evaluation

Non-strict **or** ***lazy*** **evaluation is characterised operationally as computing as little as possible to determine the result of a program. A** ***demand driven*** **evaluation policy is consistent with this idea. The notion of how little to compute is model dependent, and implementations achieve lazy evaluation to varying degrees. Nevertheless, it contrasts sharply with the more traditional computation strategy,** ***strict*** **evaluation, in which the constructs of the source language dictate when values are computed rather than relying more exclusively on data dependencies.**

In many ways lazy evaluation seems ideal for $\mathcal{PL}$, since it is consistent with the following notions:

- Delayed binding, since the decision of when to evaluate things is postponed.

- Rapid implementation, since it frees the programmer from concerns of evaluation order.
- Formal reasoning and transformation, since it allows broader and simpler forms of equational reasoning.
- Expressiveness, since it allows programming with unbounded data structures which are non-effective in other models.

On the other hand, lazy evaluation may conflict with certain other characteristics, in particular side-effects. A model with side-effects must have a notion of static sequencing in order for the programmer to control side-effects, whereas lazy evaluation results in a dynamic sequencing, one controlled only by data dependencies. We know of no general solution to this fundamental conflict, although mixed strategies have been attempted with some success, such as providing explicit sequencing constructs in an otherwise order-independent language, or defining order-independent sublanguages within otherwise strict languages. Another potential problem with lazy evaluation is that its dynamic notion of sequencing makes reasoning about *efficiency* more difficult, although efficiency in a prototype may not be an overriding concern (see the section On Efficiency below).

Because of these conflicting concerns, we make no remarks with respect to non-strict (lazy) versus strict semantics.

Requirement: ***[Lazy Compatible with Imperative]*** However, if a non-strict semantics is chosen for any component of $\mathcal{PL}$, the semantics shall be reconciled with the imperative component of the language.

Non-determinism

Requirement: ***[Non-Determinism]*** $\mathcal{PL}$ shall allow non-determinism.

Non-determinism is the ability to specify an outcome as being arbitrary over some spectrum of possibilities, as opposed to an ***angelic choice*** of the known best or correct choice from among several alternatives. That is, if a program can successfully complete only if the correct choices are made at every point of non-determinism, then we say that the choices are ***angelic*** if somehow exactly those correct choices are made. This is especially important in the context of many defense-related projects in which real-time response to non-deterministic events is common. In addition, there is an important motivation beyond these inherently non-deterministic applications: being able to express non-determinism can be a liberation from over-specification. For example, many problems are insensitive to the order certain operations are applied (the reduction of a binary associative operator over a sequence is one example), yet most programming languages require the over-specification of the order. Being able to express non-determinism allows the programmer to avoid non-essential detail, thus speeding up the development and evolution of a prototype.

Remark: ***[Explicit Non-Determinism]*** It is desirable for $\mathcal{PL}$ to allow non-determinism to be explicitly expressed.

A more difficult question is how the non-determinism should manifest itself in $\mathcal{PL}$. Since non-determinism can complicate reasoning about programs, we suggest an approach in which the non-determinism is expressed *explicitly*, since it allows isolating the non-deterministic components of the system. A language supports implicit non-determinism if non-determinism is part of the underlying computational model but no explicit diction is required to express it.

Remark: *[Non-Determinism Pragmas]* It is desirable for $\mathcal{PL}$ to support pragmas for making the implementation of non-determinism tractable.

A related issue concerns the realization of the non-determinism. If it is in the language, then implementations must support it, although it is unreasonable to expect completely fair implementations of non-determinism, especially on a sequential machine. Pragmas offer one way to make such implementations tractable; for example, a pragma could specify a randomised implementation of non-determinism that may be good enough for test purposes.

Non-effective Computations

Most programming languages are capable of *expressing* non-effective (i.e. non-computable) computations to some degree—for example, in most languages one can try testing for the equivalence of the values returned by two non-terminating programs. However, if the semantics (or computation model) for that language insists that the answer to that question be *true* (as opposed to undefined), then the language is said to be *non-effective*. Another example of possibly non-effective diction is quantification over an infinite set.

Requirement: *[No Non-Effective Constructs]* $\mathcal{PL}$ shall only employ effective constructs.

Although it is true that some specifications are non-effective, $\mathcal{PL}$ and its computation model must only employ effective constructs. In fact, we can take this to be the hallmark difference between a general specification language and $\mathcal{PL}$: specifications are not generally executable, whereas prototypes must be.

On Formality

Remark: *[Formality]* It is desirable for $\mathcal{PL}$ to be as formally defined as is reasonable.

Some prototypes will require formality in the semantics of $\mathcal{PL}$ and its computation model, others will not. A prototype acting as an executable specification or one that is expected to systematically evolve into a final product needs to have enough formal underpinnings to guarantee correctness. At the same time, these formal underpinnings should not preclude writing throw-away prototypes whose formality is not required or desired, or expository prototypes which may be sketchy and incomplete. However, although these two scenarios seem to conflict, it should be pointed out that a *formal* model does not necessarily imply a *complete* model. There may be certain semantic components that, because of other goals such as the desire for delayed binding, are purposely left undefined. Such incompleteness, if carefully specified, will not interfere with formal reasoning about those portions of the prototype that do not depend on the incompleteness.

On Efficiency

Requirement: *[Efficiency] PS* **shall be designed to have acceptable performance when implemented using existing implementation technology—a large prototype should run no worse than an order of magnitude slower than the same program in Ada.**

The various computation models and the other design considerations raised in this chapter may imply varying degrees of efficiency, in that state-of-the-art implementation techniques vary for each of them. Although the efficiency of a *PL* program should not be an overriding concern, it *can* have a significant impact on the usability of *PL*. Small constant factors in efficiency should not drive the definition of *PL*, but factors so large as to make a qualitative difference in the prototyping process must. The design choices made in *PL* must not impose so large a quantitative performance penalty as to impose a negative qualitative effect on the prototyping process.

On the Impact of Change

A final, but not insignificant, consideration is the impact a possibly unfamiliar computation model might have on the intended community of users, which in our case is primarily traditional defense contractors using Ada. For example, a model which is at odds with Ada's imperative state- and object-oriented computation model may cause problems such as the following:

- Significant training and education may be required.
- Systematic transformation (whether automated or not) *from* a *prototype* to final product may be difficult.
- The psychological transition between building a prototype and buiding a real system may be difficult, or at least awkward.

On the other hand, we should point out the following:

- There may be good technical reasons for abandoning the Ada computation model in favor of other models seen as better suited to prototyping.
- Those having experience with verification, specification, and annotation methods (such as those embodied in Anna) may not find the dichotomy bothersome.
- There may be perfectly satisfactory (and automatic) ways to transform the new computation model into more traditional ones.

We suggest that the designers of *PL* recognise that *PL* will only be successful if it is widely used by a diverse community and that abrupt changes in the way such users must think may prove an insurmountable problem.

Abstraction

Abstraction in programming is the process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use. The most familiar example of this is ***procedural*** or ***functional abstraction***, in which the common pattern is a computation which varies according to input values. The abstraction is called a *procedure*, and is applied by supplying actual input values. Abstraction yields very high code density, and provides a simple means to alter common behavior. Therefore, abstraction is a means of providing low inertia and rapid construction of prototypes.

An abstraction facilitates separation of concerns: The implementor of an abstraction can ignore the exact uses or instances of the abstraction, and the user of the abstraction can forget the details of the implementation of the abstraction, so long as the implementation fulfills its intention or specification. Creating abstractions to provide a common behavior is sometimes called "abstraction via parameterisation." Creating abstractions to hide implementational details is sometimes called "abstraction via specification."

Good abstraction mechanisms can speed the creation of prototypes, enhance the clarity of the resulting prototype, and provide a means to reuse components. Therefore, *PL* should have powerful abstraction mechanisms.

Values and Abstraction in General

Requirement: ***[First-Class Data Values]*** **All data values shall be first-class.**

We will use the term ***data value*** to refer to values that are manipulated at runtime, including procedures, functions, and, depending on choices made by the designers of *PL*, modules and types. Whatever choices the designers make, all *PL* data values must be ***first-class***. A data value is first-class when it can be passed to a procedure, returned from a procedure, or stored in data structures or persistent objects.

Remark: ***[Abstraction Genericity]*** **It is desirable for *PL* to support the abstraction of any kind of entity.**

Because one may wish to express patterns over any kind of entity, not just patterns over commands or expressions, *PL* should allow such generality. For example, one should be able to abstract declarations and L-values.

Requirement: ***[Secure Abstractions]*** ***PL*** **abstractions shall be secure.**

An abstraction *is* secure if the behavior of a program that uses it is independent of the choice of implementation of that abstraction, assuming that the alternative implementations satisfy the specification of the abstraction. An abstraction is still secure if different implementations have different performance or cannot be instrumented the same way, or if behavioral differences are the result of explicit non-determinism. Another way of expressing this concept is that abstractions

should be representation-independent.

Data and Data Abstraction

Requirement: ***[Representation-Independent Data Abstractions]*** *PL* **shall support representation-independent data abstractions.**

Requirement: ***[Enforcement of Abstraction Levels]*** *PS* **shall enforce abstraction levels: It shall enforce the constraint that values of an abstract type are accessed only through legal operations on that abstract type.**

If a data value is built using data abstraction, then there are at least two views of the value: one as a value in the *abstract type* and the other as a value in the *concrete type*. For example, if an abstraction representing queues is built using lists as the underlying representation, all instances of the queue abstraction are also instances of the type list. Typically, not every value of the concrete type is a reasonable value in the abstract type; reasonable values must pass an additional criterion, sometimes called the *concrete invariant*, which is maintained by the defined operations on the abstract type. In our example, not all lists are queues.

Remark: ***[Safe Representation Access]*** **At the same time, it is desirable for *PS* to allow the representation to be accessed when it is both safe and necessary to do so.**

This is conventionally done via an abstract data type mechanism, such as CLU clusters. *PL* should also provide for procedures that are operations on several abstract types. That is, *PL* should provide mechanisms for procedures to access the representations of several abstract types (e.g., a Common Lisp Object System generic function or a C++ function that is a *friend* of several classes).

Procedural and Object-Oriented Abstraction

Requirement: ***[Procedural/Functional Abstraction]*** *PL* **shall include a traditional mechanism for procedural or functional abstraction: that is, the abstraction of an expression over a value.**

Flexible mechanisms for passing parameters and returning values should be provided, including the return of multiple values.

Object-oriented versus Procedural

Remark: ***[Object-oriented and Procedural Abstractions]*** **It is desirable for *PL* to have both object-oriented and procedural abstraction mechanisms.**

These already co-exist in several modern programming languages. Both kinds of abstractions have their merits: an object-oriented approach distributes functionality, and a procedural approach centralises it. Having a choice for such organisational abstractions is useful.

The procedural model is well-known, but the object-oriented model has become popular only in recent years, especially in the construction of large systems. The object-oriented component should provide a mechanism for assembling generic operations, which are functional or procedural abstractions whose application may invoke different procedure bodies (sometimes called ***methods***) depending on the ***classes*** of the arguments. When the method invoked depends on the class of exactly one argument, the object-oriented model is isomorphic to SmallTalk-style message-passing; when the method invoked depends on the classes of several arguments, the object-oriented model is similar to the generic function model of the Common Lisp Object System.

Requirement: ***[Object-oriented Extensibility]*** If an object-oriented abstraction is adopted, *PL* shall allow new generic operations and new method-class associations of those operations.

Regardless of the object-oriented model selected, such extensibility is important. We explicitly make no remarks regarding whether *PL* should support SmallTalk style message-passing or the Common Lisp Object System style generic functions. The language should also include mechanisms for inheritance, that is, the definition of a new class whose behavior under the generic operations is some disciplined combination of the behavior of some previously defined classes. We explicitly make no remarks regarding whether single inheritance or multiple inheritance should be supported.

Requirement: ***[Runtime Object Instantiation]*** If an object-oriented abstraction is adopted, *PL* shall enable instances of object classes to be dynamically created.

Control Abstractions

Some programming languages provide mechanisms for control abstractions, but we were unable to come to any conclusions about whether *PL* must have such mechanisms. The following are some examples of control abstractions:

- The ability to define iterators over programmer-defined aggregations. This may be included as part of the data abstraction facility.
- The definition of sublanguages for rule-based programming.
- The definition of demon-based or event-based invocation of procedures.
- The existence of an Lisp-like ***eval*** procedure for executing data structures representing *PL* code.

Syntactic Abstraction

Remark: ***[Syntactic Abstraction]*** It is desirable for *PL* to include a syntactic abstraction facility.

Syntactic abstraction—sometimes called ***macros***—are provided by some programming languages.

In many programming languages, syntactic abstraction facilities are primitive and easily misused. Programs that use these facilities are often less perspicuous than programs that do not use them. On the other hand, within the Common Lisp community syntactic abstraction is used extensively, and there is a well-understood and controlled set of conventions for their safe and clear use. It may be worthwhile for the designers of *PL* to address the issue of syntactic abstraction.

Behavioral Abstraction and Typing

Requirement: ***[Rich Type System]*** ***PL*** **shall have a rich type system.**

A type may be regarded as an abstraction of the behavior of a data value. Types of data objects can serve as tokens that indicate their acceptability as potential parameters to or results of procedures; procedures, in turn, have types that indicate the types of their parameters and results. For example, a function might have a type notated as follows:

$$\text{list(int)} \times (\text{int} \rightarrow \text{bool}) \rightarrow \text{list(bool)}$$

This specifies that the function takes a list of integers and a function that maps integers to booleans, and returns a list of booleans.

The type of an abstraction is a necessary component of its specification. A type system in which a rich set of behaviors is expressible is an important aid to the effective use of abstraction.

Remark: ***[Typed Values]*** It is desirable for every data value in *PL* should have a type, and for that type to be available at runtime.

Requirement: ***[Strong Typing]*** ***PL*** **shall be strongly typed: that is,** *PL* should ensure that every data value is used only in ways that are consistent with its type.

This may involve runtime checks. *PL* should, however, use static typechecking to the extent that early binding of types permits, that is, the static typechecking system should not force the premature binding for the types of values and program fragments.

The static typechecking algorithm must be sound: that is, if a program phrase is assigned a certain type, then the value of the phrase must have behavior consistent with that type. The *PL* typechecking algorithm should use type inference to allow the programmer to avoid writing deducible type information whenever possible; it should, however, always be possible to include type declarations for redundancy.

Remark: ***[Dynamic Typechecking]*** It is desirable for static typechecking to not be used to eliminate runtime or load-time typechecking except in situations where performance would be prohibitively poor.

The goal of low inertia implies that early commitments be quickly and easily revocable, and this can be best if re-compilation and re-typechecking of unchanged code is unnecessary.

Requirement: ***[Polymorphic Types]*** The *PL* type system shall allow the specification of

polymorphic types. That is, it shall be possible for procedural abstractions to be applied to parameters of varying types, and it shall be possible to describe this application in the type system.

For example, one should be able to specify a procedure with the following type:

$$\forall \quad T, U \quad \text{list}(T) \times (T \rightarrow U) \rightarrow \text{list}(U)$$

Whenever possible, the type inference algorithm should yield the most general such type for any piece of code.

The type system should have a smooth interface with the data abstraction facility, and also with the inheritance system.

Modularity, Scalability, and Libraries

PL is to be used for developing both small- and large-scale prototypes. As with any large-scale program, a large-scale prototype will be dominated by issues of communication between sections of the program.

Requirement: ***[Modules]*** *PS* shall include a module system that allows the aggregation of abstractions and other program components into larger groups.

The module system should include convenient mechanisms for assembling program fragments into packages, for indexing and retrieving packages and their components, and for assembling programs from the retrieved components.

Requirement: ***[Module Specification Language]*** *PL* shall include a module specification language.

A module is itself an abstraction that encapsulates its contents. Therefore, *PL* must provide a module specification language for describing the contents of modules. The module specification language should allow the description of the names of the objects in the module, their types, and the permitted operations on the objects. The module specification language may also describe other components which may be utilized by the library and reuse system.

Remark: ***[Module Indexing System]*** It is desirable for *PS* to provide some mechanism for indexing a library of modules.

A library of modules is of limited utility unless there is a useful indexing system, ideally one based on semantic content of the modules in the library (though such systems require further research). We believe that facilities for well-cataloged libraries will be an important adjunct to *PS*.

Scope

Wide-spectrum Language

Requirement: *[Wide-Spectrum] PL* **shall support both high and low level constructs.**

To allow both behavioral and structural prototypes, *PL* must allow a wide range of styles in specifying computations, which might loosely be characterised as ranging from very high level to much lower level. The following are some examples of points along the spectrum that might be useful in a prototyping context:

- *PL* should permit high level functional style programming at one extreme, and the introduction of variables and imperative notions at the other extreme.
- *PL* should permit high level data structures such as sets and maps, but should also accommodate lower level notions such as pointers and arrays.
- Storage allocation and layout should be elided entirely at a high semantic level, but *PL* should be able to accommodate explicit consideration of storage allocation and mapping of data onto storage units.

Requirement: *[Expressiveness] PL* **shall consistently incorporate the highest level, most expressive constructs that remain effective.**

Mathematics represents an idealised extreme in which effectiveness is not required, but is also offers a rich body of highly expressive and effective notions.

Requirement: *[System Tools] PL* **shall be capable of constructing system tools and components, in particular portions of** *PS* **itself.**

This implies the capability in *PL* to treat types and abstractions interpretively as runtime bindings so that *PS* tools (such as debuggers and information extractors) can operate on the types and abstractions contained in user prototypes.

It is not required that *PS* be constructed in itself, at least initially.

Real-time and Concurrent Computation

Requirement: *[Real-Time] PS* **shall support the prototyping of real-time, concurrent applications.**

We recognise that it is difficult and probably impractical to capture all aspects of real-time considerations, such as determining whether a prototype meets its deadlines. A high level prototype by its nature is unlikely to be able to model at a sufficiently low level to provide meaningful data on such issues. Prototypes that directly reflect low level real-time concerns possibly can be written only at the same level as its final implementation, and probably only by using the production

compiler and target system environment. The prototype may in this case be able to omit many of the eventual details. However, *PL* is by its nature not intended to encourage prototyping at a low level and might not meet this need.

On the other hand, it is certainly possible to model the logical aspects of a real-time system, including such things as division of the work among separated tasks and the communication between such tasks at a high level. To achieve this, *PL* must contain appropriate linguistic features to represent concurrency. Furthermore, these must bear a reasonable resemblance to the features of the target implementation language. One simple approach is simply to graft the tasking semantics of Ada onto *PL*. It should be possible to do better than this and find higher level constructs that can be used to model multiple task applications, but which nevertheless can be appropriately mapped onto the lower level features of the target language in the case where the prototype is used as a model for the eventual implementation.

It should also be noted that even in applications where concurrent use of multiple tasks is not expected in the final application, it is often useful to model interaction between separate parts of a high level solution using concurrent models. For example, in a complex office control system, a high level prototype might use separate tasks to represent different functions to be performed, interacting in specified manners, while the final low level implementation might be expected to be implemented as a single process reading a queue of operations to be performed.

The following are typical questions that might be asked of a prototype modeling real-time and concurrent processes:

- How much task switching overhead is present?
- Do all cyclic tasks meet their deadlines?
- Are the priorities assigned optimal?
- How much spare CPU capacity is available at various cyclic frequencies? On average? In the worst case?
- Are there any time-dependent race conditions?
- What system clock frequency is needed for adequate response?
- Is an event-based scheduler or a priority scheduler the appropriate approach?
- If a particular scheduling policy is used (e.g. rate monotonic scheduling), how close is the system performance to that predicted by theory?
- Is the handling of specialised devices correct?
- What are the worst case hardware interrupt handling situations? Are interrupts missed? With what frequency? Does the program handle missed interrupts gracefully?

It is unlikely that prototypes written in *PL* will be able to provide answers to these questions, and it is ambitious to expect that *PS* will be initially powerful enough to provide these answers

from prototypes put together using target language components. We would be pleased to see such a *PS*.

Parallel Computation

It is likely that an increasing number of computation problems will be addressed using specialised parallel architectures. The possible range of such architectures is very wide. *PS* should accommodate writing prototypes for such systems.

Remark: ***[Implemented on Parallel Architectures]*** It is desirable for *PL* to be able to be effectively implemented using parallel architectures.

One of the important approaches in prototyping is to run the prototype on a fast machine to make up for the inefficiency of the prototyping approach. In particular, taking advantage of highly parallel machines is an attractive option for running prototypes at an acceptable level of efficiency.

Remark: ***[Prototyping Parallel Architectures]*** It is desirable for *PL* programs to be able to model computations performed on a wide variety of parallel architectures.

In other words it should be possible to write an operable *PL* prototype that acts as an appropriate implementation specification for later efficient implementation on a specialised parallel architecture.

Remark: ***[Implementation Architecture]*** It is desirable for *PS* to be designed to run on a network of heterogeneous parallel processors.

Although such architectures are not prevalent today, we believe that it would be wise to anticipate movement in this direction. *PS* should also be suitable for degenerate systems from this class, including distributed uniprocessors, various parallel processors, and single uniprocessors. We are aware that this is ambitious, but nevertheless suggest that the designers consider this option.

Parallel Computation Discussion

We recognise that this is a difficult problem, because of the wide variety of possible architectures, but it is important that *PS* be as flexible as possible in this regard. In particular, *PS* should have features for the following:

- Running large numbers of tasks in parallel
- Establishing restricted models of communication between parallel tasks
- Operating with shared memory or local memory models
- Modeling both SIMD and MIMD computation.

Distributed Computation

Requirement: ***[Distributed] PL*** **shall support the prototyping of distributed applications.**

The use of distributed computation can also be expected to become more important, and *PS* must be able to model distributed computations effectively. Among consideration in this area are the following:

- Separate tasks may have quite different capabilities, corresponding to execution on heterogeneous processors.
- Communication between tasks needs to be able to reflect realistic communication across networks, including simulation of network traffic flow, and communications failure.
- Such techniques as fault tolerance and redundant processing should be able to be specified and prototyped.

These are somewhat open-ended, and it is unlikely that all these considerations can be accommodated, but the design of *PS* should make an attempt to allow effective and realistic prototyping of distributed systems at a high level. Of course it is not intended that *PL* necessarily have primitives to directly support these features, but the abstraction facilities should be capable of being used to build the necessary primitives.

Knowledge-based

Requirement: ***[Knowledge-based] PS*** **shall support the prototyping of knowledge-based applications.**

To date, knowledge-based applications have been poorly supported in conventional programming languages. Most such applications have been built in Lisp, Prolog, shells built upon them, or in other special purpose languages. Recently, interest has been expressed for supporting knowledge-based applications in Ada.

Most of the capabilities needed to support knowledge-based applications have already arisen as *PL* requirements from other considerations. The major exceptions are inferencing, rules, constraints, and contexts.

How such capabilities should be supported is still an open issue. We therefore do not require any specific *PL* language capabilities, nor even that knowledge-based applications be directly supported by *PL* constructs rather than indirectly by a library of modules. Instead, we merely require that *PS* somehow facilitate the prototyping of knowledge-based applications. However, we feel that the designers of *PS* should consider providing explicit mechanisms to support some or all of the following:

- Representation primitives for key knowledge programming types, including facts, rules, imperative blocks, classes, instances, type hierarchies, frames (structures, inheritance and demons), predicates, constraints, instance generators, patterns, and contexts.

- Representation for key meta-knowledge types: logical moods (e.g, certainty, plausibility, conditionalities), applicability restrictions or conditions, authorship, and priority.
- Efficient and general inference engines for pattern matching, unification, data-directed forward chaining, goal-directed backchaining.
- Aids for incremental knowledge changes, including automated testing and semi-automated configuration management.
- Capability for quickly building application-specific knowledge representation, inference, instrumentation, and management capabilities.
- Capability for modifying built-in knowledge programming capabilities and for dropping out capabilities or adding new-ones.

Syntax

The following three issues are to be addressed:

- Should *PL* have a specified abstract syntax? If so, how should it be defined?
- Should *PL* have one or more specified surface or concrete syntaxes, or should the concrete syntax be flexible? CIP-L is an example of a language which has taken the latter approach.
- If there is a specified concrete syntax, what general style is desirable?

Concrete Syntax

Requirement: *[Single Concrete Syntax]* ***PL*** **shall have single a well-defined concrete syntax.**

Since *PL* is a notation for communication of ideas between people, as well as communicating with machines, it is important that everyone adopt a similar, mutually comprehensible style of programming, and using the same syntax is a required component of such similarity. In practice, either by fiat (as in COBOL), by suggestion (as in Ada), or simply by custom (as in C), the commonality extends beyond the syntax issues per se and includes such issues as layout and naming. The question of the extent to which the language definition should address such subsidiary questions is left open.

Another reason for this requirement relates to tools. Although many tools will operate at the abstract syntax level, there will be a number of tools that legitimately will work at the concrete syntax level, including for example reformatting tools and source maintenance tools, and possibly *PL* compilers. It is important that such tools be fully portable and usable in all contexts. Requiring such tools to be parametrised by syntax definition would considerably complicate them and in some cases be impractical.

Concrete Syntax Style

Given the requirement that $\mathcal{PL}$ should have a primary concrete syntax, the issue is what this syntax should look like. In designing this syntax, the general aim should be to stress the coupling between language and environment, and to improve the analysability and readability of the program.

This leads in the direction of a publication type syntax which takes maximum advantage of specialised notations such as set former notation, as in the following example:

$$\{f(a) \mid a \in A\}$$

Such a publication syntax is intended for communication of $\mathcal{PL}$ programs to readers, although $\mathcal{PL}$ processors should support this syntax fully.

The $\mathcal{PL}$ design should deal with the issue of usability of specialised notations that are incorporated into the syntax, for example, with notations for the design of specialised display and printer fonts, along with convenient tools for inputting programs in this form.

Abstract Syntax

Requirement: *[Single Abstract Syntax]* $\mathcal{PL}$ shall have a single well-defined abstract syntax.

Part of the $\mathcal{PL}$ definition is a standardised abstract syntax, which will be used by all $\mathcal{PL}$ compilers and other tools. By standardising an abstract syntax, $\mathcal{PL}$ tool components will be more interchangeable, and such tools as pretty printers, cross-reference, and other analysis tools using the abstract syntax will be portable across implementations.

The abstract syntax should be in the form of an attributed tree. The extent of the attribution may vary depending on the use of the syntactic object. For example, if $\mathcal{PL}$ includes some kind of type inference mechanism, then in one view, perhaps prior to this inference operation, the tree would not contain type annotations, but it would contain such annotations after type inference. The abstract syntax tree should be representable in the following two manners:

- As an abstract data type, using the standard abstraction mechanisms of $\mathcal{PL}$, so that it can be manipulated as a data object in $\mathcal{PS}$. In particular, $\mathcal{PL}$ programs may create other programs or program fragments in this form. Since the abstraction mechanisms of $\mathcal{PL}$ provide for multiple views of abstract types, the various levels of detail, corresponding to different sets of attributes, can be represented using this mechanism
- In character string form, so that abstract syntax trees (AST's) can be transferred across the boundaries of the $\mathcal{PL}$ system in a form conducive to easy transfer by existing mechanisms such as mail messages and magnetic tapes. This allows transfer of AST's from one $\mathcal{PS}$ to another one which uses different internal representations for AST's.

  At the simplest level of abstraction, the character string representation of a $\mathcal{PL}$ AST might

look very similar to Lisp.

Tools for converting from the concrete syntax to AST's (translators) and vice versa (pretty printers), can be standardised across different instances of *PS*.

Note that since tools exist for dealing directly with AST's expressed in character form, it is possible that this form could be used directly by other non-*PL* programs or by programmers in communicating *PL* texts to the system. The extent to which similar usage patterns in *PL* will depend both on the design of concrete syntax, and on the habits of programmers using the system.

Syntax Discussion

We considered the idea of *PL* specifying an abstract syntax and providing mechanisms for easily describing concrete syntax as a customisation of *PL*. This idea was rejected for several reasons.

CIP-L is a language that adopted this approach. In the case of CIP-L, what happened in practice was that three variants were developed—an ALGOL variant, a Pascal variant, and a Lisp variant. The first two were essentially identical, the main issue distinguishing them was the hotly argued difference over the use of semicolons. The Lisp variant has been used much less. Many people connected to the project seemed to indicate that the original decision was made for essentially political reasons, and that in retrospect it made little technical sense.

On the other hand, the Working Group recognises that the experience of such a small community as the CIP-L community possibly does not apply directly to this situation.

If there were no defined syntax, then standards would certainly develop, but the development would be haphasard, compared to the advantage of initial standardisation. Syntax is something well understood, so there is no real merit in deferring decisions on the concrete syntax.

One idea we considered was that all communication between different users of *PL* be at the abstract syntax level. This would work fine for communication between *PL* processors, but it would not work for communication between people in the Ada community. Even in the case of communication between processors, lack of syntax standardisation means that effort is wasted on generating multiple front ends which are not sufficiently strongly needed.

A system is weak whose main feature is that it is customisable without providing a recognisably useful set of defaults. Acceptance of *PS* will be hindered or severely delayed if a lengthy period of customisation and default standardisation is required.

All these considerations add up to suggesting that the *PL* design should include a standardised concrete syntax.

Target Language Modeling

Behavioral Prototypes of Target Components

Requirement: ***[Behavioral Prototypes]*** *PS* **shall provide the ability to develop behavioral prototypes of target language components.**

Either through capabilities provided in *PL* itself or through the interoperability capabilities, the entire functionality of the target language must be available to the behavioral prototype developer. It is desired, but not required, that *PL* itself have the capability to be at least as expressive as Ada, mainly through suitable abstraction capabilities. It is up to the designers of *PL* to decide how much of the target language to model in *PL* and how much to provide through interoperability.

In contexts where there is substantial ongoing development in a target language, this might be a typical use for *PL*. It is hard to imagine that *PL* would be acceptable to a large target community without satisfying this requirement.

Structural Prototypes of Target Components

Requirement: ***[Structural Prototypes]*** *PL* **shall provide the ability to develop structural prototypes of target language components.**

In target languages such as Ada which has clear aggregation units or component modules the structure of the program is largely revealed in the module selection and the interrelationships among the modules; *PL* ought to be capable of modeling that component structure, including the relevant aspects of the module interfaces. Any of the component modules may be modeled by a behavioral prototype, so structural prototypes must be able to incorporate behavioral prototypes.

In contexts where there is substantial ongoing development in a target language, this is another of the most typical uses for *PL*. It is hard to imagine that *PL* would be acceptable to the Ada target community without satisfying this requirement.

User Interface Support

Requirement: ***[User Interface Prototyping]*** *PS* **shall support the rapid prototyping of user interfaces.**

One of the characteristics of modern computing environments is a much more sophisticated input/output environment, including the use of high resolution graphical devices, windowing systems, and laser output devices. We feel that many prototypes will be of systems that make use of these devices. Of particular importance is the ability to support rapid development of user

interfaces. Because there is a trend towards designing the user interface before designing the rest of the system, user interface support is extremely important. Requirements engineering often entails providing a user interface that can be tried out by a sample of users.

We note this particular point because few language designers focus on user interface issues, leaving that up to the environment or a library.

3. An Environment

Introduction

$\mathcal{PE}$ is a set of tools for helping a prototyper get a prototype written and running. In general, the more advanced the prototyping environment the more open and scalable it is, and the more integrated are its tools. By *integration* we mean that the tools are interconnected in such a way that moving from tool to tool during a prototyping session is easy.

As with $\mathcal{PL}$, $\mathcal{PE}$ has both a component which it shares with programming systems, and an additional prototyping-specific component that is unique.

This chapter identifies the requirements pertaining to these portions of $\mathcal{PS}$.

Programming Environment

There are two distinct parts to the generic programming component of $\mathcal{PE}$. The first is the *runtime system* which supports the semantics of $\mathcal{PL}$. It performs memory management, ensures runtime type consistency, provides linkage among procedures and functions written in supported languages, and supports instrumentation and monitoring. The second is the development environment, which provides programming tools like editors, debuggers, and browsers, along with the user interface, the dynamic loader, the window system, and presentation tools for instrumentation and monitoring.

Requirement: *[Modern Baseline]* $\mathcal{PE}$ shall encompass the most up-to-date generic programming environment techniques.

Modern programming environments are window-based, dynamic, incremental, and well integrated. They contain many capabilities detailed below that are needed for both programming and prototyping. Several existing systems, including the Lisp Machine, SmallTalk, APL, Cedar, and Rational Computer's Ada illustrate the current state-of-the-art.

Remark: *[Existing Platform]* It is desirable for these and other similar programming environments to be considered as an implementation base upon which $\mathcal{PE}$ is constructed.

These environments represent enormous implementation investments, usually several times that required for the language they support. A significant proportion of the implementation effort goes into engineering these capabilities into a smooth, responsive, and coherent system. Because they contain so many capabilities in a single well-engineered, integrated system they offer the potential of a high leverage platform upon which further capabilities can be built.

With respect to the generic capabilities, major additions to these existing systems are needed to support a persistent object base, multiple versions of prototypes (both chronological and variants), and multiple users.

In the rest of this section we identify some of the tools already part of these advanced programming environments and which must also be included in *PE*.

Editor

Requirement: ***[Editor]*** ***PE*** **shall contain an modern screen-based editor.**

A programmer interacts with a computer through the keyboard, the screen, and the mouse. An editor is a program that manipulates text and other material on the screen using the keyboard and the mouse.

A good editor will be engineered to aid the programmer with the mechanical tasks of entering his program. The editor should be able to check syntax, to determine well-formedness of programs, to detect type errors, and to check that the module structure of the program is consistently followed. That is, there is no reason that the editor should not be able to handle the actions normally associated with the syntactic and semantic phases of the compiler.

The syntax checking performed by the editor should be unobtrusive.

System Definition Tools

Requirement: ***[System Definition]*** ***PE*** **shall contain a facility for defining systems, for locating the various versions of the components they contain, and for rebuilding it after modification.**

Many modern programming environments provide a module and library facility, but beyond that facility is the need for mechanisms to enable the programmer to find the definitions of routines and data structures within those modules and libraries. When a programmer wants to look at some definition he does not want to spend time and attention locating that definition. Keeping track of such definitions is something that the programming environment should be able to easily do.

When some definitions are altered, the programmer will want to have those changes incorporated in his running system. A good system definition tool will be able to determine the quickest route to that situation. Only under the most extreme situations will it be necessary to re-compile or re-load the system.

Supporting multiple versions of parts of some modules helps achieve the goal of low inertia—changes to the prototype can be quickly and easily made. This facility might be supported by *PL*, but it is typical for such things to be supported by the environment. The programmer should be able to write special compilation or redefinition protocols using this tool. That is, if there is some means for the programmer to specially compile or load some part of his program, it ought to be possible for him to write a *PL* program to do so. This is one example of utility of the requirement that *PL* be capable of expressing *PS* tools.

Debugging

Requirement: ***[Debugging]*** *PE* **shall provide a modern source level debugger which is capable of interactively evaluating arbitrary** *PL* **expressions or statements in the current context, modifying bindings and values within the prototype, and alterning the point at which an interrupted computation will continue.**

Unfortunately programmers make mistakes. In this case it is often useful to be able to examine the state of affairs while in the erroneous situation, possibly using a *PL* debugger. A common style of debugger uses a ***listener***, which is a loop that reads expressions, evaluates them in the current environment, and prints the results.

Other actions that are useful while debugging are to invoke an ***inspector*** to look at data structures, to look at the invocation stack to see what procedure calls led to the current situation, to examine the call frames for each procedure invocation, and to rewrite and re-execute code. Some debuggers provide mechanisms to enter the editor and engage in a separate debugging session.

A very powerful tool for debugging is a ***single stepper***. A single stepper executes source code a statement at a time, pausing after each statement execution to accept user commands.

A related tool provides a mechanism to run programs at less than full speed but faster than single stepping. Such a tool used in conjunction with the monitoring tools is useful for viewably displaying critical parameters or some representation of the flow of control.

The *most effective debuggers* operate in environments that are fully ***instrumented***. An instrumented environment is one in which the types of objects are apparent, and procedures and access to variables may be monitored. See the description of Information Extraction in the Prototyping-Specific Capabilities portion of this chapter.

Browsing

Requirement: ***[Browsing]*** *PE* **shall allow all values to be browsed and edited.**

Browsing **is the activity of examining data structures, variables, and procedures. A variety of presentation techniques, each tailored for the display of certain sorts of information is often most effective. For example, a program might be displayed as a graph of procedures with data-sharing relationships highlighted. Another example is hiding comments and documentation when browsing program text, or vice versa.**

It is useful for anything that can be browsed to be also editable. This may require a set of editors, each designed for editing a particular type of object using a particular presentation.

Dynamic Loading

Requirement: ***[Dynamic Loading]*** *PE* **shall support the dynamic linking of separately compiled modules or procedures.**

One goal of the *PS* is to be able to trade efficiency of programs for efficiency of programmers. One means to gain programmer efficiency is to speed up system turnaround. A dynamic loader is a tool for incrementally recompiling and loading single procedures (or even single lines from a procedure, in some cases). Such a tool requires that the usual static linking of code be abandoned in favor of dynamic linking; however, there are several techniques for minimizing the cost of dynamic linking at runtime. As with all prototyping environment tools, safety and consistency checking should be performed by default when dynamically linking programs.

Windowing

The need for a window interface is apparent for several reasons. A window interface can be used to present information to a user in an efficient manner and, if it is flexible enough, it can be used to prototype user interfaces. The first of these is treated here and the second is addressed within the Prototyping-Specific Capabilities portion of this chapter.

Requirement: ***[Window Based User Interface]*** *PE* **shall have a modern window based user interface.**

The presentation of monitoring and other information as part of *PS* is best accomplished with a window system. With high resolution graphics devices, the prototyping environment can be written to display more information than with other devices, and the displays can be natural for the task.

Multitasking

Requirement: ***[Multitasking]*** *PE* **shall support multitasking within** *PS*.

Multitasking is useful to have in *PS* because a programming environment is most flexible when the programmer is able to work on one task while the system is able to complete some other tasks.

Multitasking is usually required to implement a window system. For example, one process is used to poll the mouse in order to implement mouse-sensitive regions of the screen for menus.

PL **is also required to support multitasking, and it is reasonable, but not required, for** *PS* **and** *PL* **to utilise the same facility.**

Condition Signaling

Requirement: ***[Condition Signaling]*** *PE* **shall support the handling of conditions signaled by either** *PL* **components or the target language systems with which they are linked.**

Often a program will cause foreseeable categories of errors or conditions to occur at unpredictable times. The programmer will often wish to handle these errors or conditions by supplying code for

each type of error or condition. When an error of a certain type occurs, the associated handler will be invoked. Experience with Lisp and CLU have shown that it is best for condition handlers to define a lexical scope and for the condition handlers to be in force during the execution of the lexically enclosed expressions.

Often a prototype's behavior can be well-specified by specifying the normal behavior along with the exceptional behavior, and in this case the entire prototype could be well-specified by *PL* code along with condition handlers.

Integration of Tools

Requirement: ***[Tool Integration]*** **All *PE* tools shall be integrated and operate on mutually understandable representations.**

The editor should know about the syntax and some of the semantic rules of *PL*, and possibly of the target language; the compiler, the profiler and the debugger should know about the editor, for example. Errors made during the coding phase of prototyping should be caught as early as possible and presented in the most useful contexts.

Debugging, monitoring, and other extracted information should be associated with the source code affected. If a performance profiler determines the relative frequency of the various arms of a conditional branch, that information should be integrated with the source for that conditional.

Similarly, debugging information should be displayed with the relevant pieces of code. The display of such information should be uniform, but customisable by the prototyper.

Extensibility

Remark: ***[Extensibility]*** **It is desirable for *PE* to allow prototypers to extend (their version of) *PS*.**

Two philosophies exist concerning the extensibility of *PS*. One philosophy is that languages and systems are designed by language designers and system designers, and a user is not in a position to make good decisions: any decisions made about language design made by someone other than the language designer will lead to non-portable or poor code.

The other philosophy is that there should be no distinction between a designer and a user, and that anything the implementor of a system wrote could be written by a prototyper.

We feel that *PS* should allow prototyper extensibility. Any organisation that ascribes to the other philosophy can adopt a policy prohibiting such extensions.

Prototyping-Specific Environment Features

The programming environment features discussed so far are simply tools that programmers expect in their environments, though it is sadly the case that many programmers do not routinely enjoy the use of such environments.

In addition to these capabilities, $\mathcal{PE}$ needs prototyping-specific capabilities for extracting information from a prototype, running an incomplete prototype, redefining portions of it, rapidly prototyping user interfaces, and for linking a prototype with the rest of the software lifecylce. These issues are the focus of this section.

Information Extraction

Extracting information is one of the two activities that distinguishes prototyping from programming. This information is used to show what, how, and how well the prototype is doing. It can also be used to determine whether the prototype is functioning as intended and, if not, why not. That is, this information is used both for testing and debugging. Supporting information extraction requires mechanisms to do the following:

- building test harnesses needed for tests
- constructing scenarios to drive tests
- writing test evaluators
- reinitialising the state of an executing prototype in preparation for a test

We divide information extraction into three areas: gathering the information (Instrumentation), determining whether the prototype is functioning as intended (Testing), and making the information visible to the user (Presentation).

Instrumentation

It must be possible within $\mathcal{PE}$ to capture, manipulate, and aggregate any instrumentation data desired. It must also be possible to to compose both built-in and user-defined higher level instrumentation capabilities. Examples of necessary instrumentation and uses of instrumentation are as follows:

- Execution times and counts for code
- Execution counts for data structure access (separate reads and writes)
- User interaction characteristics (for recreating user interaction situations)
- Comparison of several prototype executions with each other.

Requirement: *[Placement of Instrumentation]* It shall be possible to instrument any part of

a prototype (e.g. procedures or types), its data, or any of the control or data abstractions upon which they are based. It shall be possible to invoke a *PL* or target language procedure whenever a instrumentable part of a prototype is accessed.

We feel that it should be possible to easily attach instrumentation to any object that can be instrumented for the purpose of gathering information. It should be possible for attached instrumentation to include code to be activated each time control or data passes through the item to which it is attached. For abstractions, this should happen each time control or data passes through any instantiation of the abstraction. For the *process* abstraction, this should happen when the process starts and stops. Upon each activation, the attached instrumentation should have access to all the relevant aspects of the operation occurring (e.g. whether the data operation is a read or a write along with the old and new values, the actual arguments of a routine being invoked, the value being returned, etc.), and it must be possible for the attached instrumentation to time the operation. For data, this should happen whenever the datum is accessed, and the attached instrumentation should have access to the type of operation and whether it is a read or a write.

Requirement: *[Data Aggregation]* It shall be possible, using *PL* , to conditionalise and filter the data gathered during instrumentation and to aggregate instrumentation data.

Using instrumentation and *PL* composition, we expect designers to be able to satisfy the following additional higher-level instrumentation requirements:

Requirement: *[Instrumentation Specification]* It shall be possible within *PL* to monitor changes to data satisfying a *PL* predicate.

Using whatever means *PL* provides for expressing predicates and aggregating data (such as set formers), it must be possible to specify the instrumentation to be attached to each element of an aggregation.

This capability raises a design issue. Since instrumentation has access to both old and new values when data is being updated, it is reasonable to allow either to be used in determining which changes to monitor. However, this requires that such two-state access be part of the *PL* predicate language, that the predicate language be specially augmented for use in instrumentation, or that the filtering occur within the computational portion part of instrumentation. We believe that this choice should be made by the designers of *PL*.

Requirement: *[Instrumentation Composition]* Prototypers shall be able to define compound events in terms of the activating operations and/or other previously defined compound events, and to ask for and see instrumentation in terms of those compound events.

Requirement: *[Low Instrumentation Inertia]* Prototypers shall be able to quickly and easily turn instrumentation on/off or to change what is instrumented.

This requirement could be satisfied by making instrumentation a *PL* object that is manipulated by higher-level *PS* tools.

Testing

Testing is the determination of whether a prototype is functioning as intended. This determination requires the comparison of actual versus intended data. The data can include both the behavior of the prototype and its performance, and the behavioral data can include data from the final state of the prototype as well as from any of its intermediate states.

If the determination is being performed by some tool, then the intended data (or some characterisation of it) must be formally represented so that a comparison can be made. If the determination is being performed by the user, then the actual data must be presented (i.e. shown visually) so that the user can assimilate this data and compare it with the intended data which may be implicit or described informally.

Requirement: ***[Query Access]*** Prototypers shall be able to access (via query) all data satisfying a $\mathcal{PL}$ predicate, with some exceptions.

Since the determination of whether a prototype is functioning as intended by a user is informal, prototypers often formulate additional queries on the basis of what has been seen so far. This interactive ability should exist while in the midst of execution. That is, the prototype should not be recompiled once the predicate is known. Rather, the new data must be accessible as it then exists. To achieve this would imply the capability of being able to access all instances of a type. On the other hand, to provide this capability might incur an excessive performance penalty on $\mathcal{PE}$. Therefore, it is acceptable for the designers of $\mathcal{PE}$ to except certain types of data or certain types of predicate.

In addition to the above information extraction requirements, several more arise from the need to define and run tests, initialise the environment beforehand, reset it afterwards, and determine the success or failure of the results.

Requirement: ***[Test Initialisation]*** $\mathcal{PE}$ shall provide a low inertia mechanism to set up the environment for a test and to reset it after running that test.

Requirement: ***[Test Harness]*** Prototypers shall be able to specify the scenario for a test and to run the test according to that scenario.

This includes the ability to externally control all inputs and outputs of the prototype, supplying selected inputs from the scenario or interactively and gathering selected outputs. Also required is the ability to detect and respond to any exceptions raised during the the test.

Requirement: ***[Test Determination]*** Prototypers shall be able to define the criteria for success or failure of a test, and to determine which occurred.

This necessitates the ability to specify a set of expected results (which may involve intermediate values, chronological restrictions among them, and performance measures) and the means of comparing them against the set of actual results (gathered by instrumentation). This requirement would be easily satisfied if these computations could be written in $\mathcal{PL}$ and appropriately invoked.

Remark: ***[Non-Angelic Constraints]*** It is desirable for prototypers to be able to define conditions that should not be violated during the execution of the prototype.

The system would have the responsibility for detecting any violation of the stated conditions.

Such a capability would be very useful in testing. Existing testing mechanisms examine behavior and compare it against expected behavior. Non-angelic constraints would provide a means of identifying undesired behavior at a fine level of granularity.

There are at least the following four possibilities for what can happen when a constraint is violated by the prototype:

- ***Break:*** Stop and interact with the user.
- ***Instrument:*** Record the violation and continue.
- ***Constraint Propagation:*** Repair the violation and continue.
- ***Error:*** Raise an exception.

One question not adequately addressed by the Working Group is whether such constraints should become part of $\mathcal{PE}$. Their inclusion would certainly raise the level of the prototyping language by introducing a powerful declarative construct—the continuously evaluating violation condition. The manner in which constraint violations can be handled also alters the tenor of $\mathcal{PE}$ in ways not explored by the Working Group. A final issue concerns the performance implications of such a capability.

Presentation

Making the collected data visible to the user in a useful manner is imperative. Furthermore, it is important that the data be stored in such a manner that it is subject to analysis programs. Some data is best displayed graphically, such as the relative execution times of portions of the prototype. Other data is best displayed as an annotation to the source code, such as the relative frequencies of execution of different parts of the code.

Other forms of integrated presentation would make prototyping easier. Examples are the display of performance information as part of a display of the static call tree, or the display of access counts or percentages as part of the display of data structures.

The Working Group does not have specific remarks concerning how this part of the user interface should be designed.

Incomplete Prototypes

Requirement: ***[Run Incomplete Prototypes]*** $\mathcal{PE}$ **shall allow incomplete prototypes to be run.**

A prototype is incomplete if not all procedures, functions, or types are defined or if they are partially defined.

$\mathcal{PE}$ **must be able to run incomplete prototypes up to the point where such definitions are required.**

At this point, a surrogate can be invoked, a condition can be signaled, the user can be queried, or any of a variety of actions can take place.

Incremental Development

Prototyping is necessarily incremental. *PE* should support this by providing incremental compilation for *PL* code and dynamic loaders for *PL* and target components. Automatically generated condition handlers should be used to handle stubbing and other problems introduced by the requirement for incomplete prototypes. Multiple execution spaces might be a reasonable means for rapidly debugging prototypes.

Requirement: *[Redefinition]* *PE* shall allow all *PL* definitions to be modified and reinstalled.

One aspect of the incremental nature of prototype development is the need to redefine previously defined procedures, data structures, classes, types, and protocols. Such redefinitions must be allowed but once they have been used to create instances, an issue arises about what to do with those instances of the (now) obsolete definition.

A variety of approaches are possible. The one used by the Common Lisp Object System is to provide a means for the prototyper to specialise a system-supplied *update protocol* in which instances of redefined objects are modified to correspond to the new definition. Implementationally, updating happens lasily, that is, when they are accessed.

Remark: *[Instance Update]* It is desirable for *PE* to provide a mechanism like the update protocol to update instances of redefined objects.

Prototyping User Interfaces

Requirement: *[User Interface Prototyping]* *PE* shall facilitate the prototyping of user interfaces.

The window interface can be used to supply modules for prototyping user interfaces or for the user interface portions of prototypes. Often the most time-consuming work during prototyping is writing the user interface. Because *PS* ought to help make prototyping faster and easier than programming, the availability of a set of tools and modules is important.

Remark: *[User Interface Prototyping Toolkit]* It is desirable for *PE* to provide a user interface toolkit to facilitate user interface prototyping.

Currently there are available user interface toolkits that are designed to help programmers design user interfaces by providing mechanisms to select commonly used components and customise them, to draw graphical components, to render components active to mouse motion and input, to compose these components into larger ones, to provide animation and audio feedback, and to provide various means of input to the program.

Lifecycle Support

If our hopes about the software lifecycle are fulfilled, prototypes will continue to exist beyond their use in the specification phase. In this case, all documentation, specification, and informal data that is derived or produced during the early phases of prototyping will need to continue to exist for long periods of time. In addition, the use of prototyping throughout the software lifecycle will be facilitated if there are tools that assist in maintaining the linkages between changes in the production implementation and the prototype.

It should be possible to easily move between source code to the corresponding documentation and specification. Because we expect that many prototypers will be developing prototypes at the same time that documentation and specification are being developed, $\mathcal{PE}$ should support this activity as part of the software lifecycle.

During exploratory prototyping and debugging, information that is neither documentation nor specification is often derived. We call this *informal* data. Insofar as informal data is associated with $\mathcal{PL}$ code, $\mathcal{PE}$ should support the management of this data alongside those $\mathcal{PL}$ structures with which it belongs.

Design decisions, structuring decisions, tradeoffs, requirements, partial specifications, comments, and hints for future programmers are all examples of what we call informal data.

A well-developed prototyping environment will support management planning and control capabilities. In some cases there are existing systems that support these activities, and it would be desirable for $\mathcal{PE}$ to interface to them.

Support for Multiple Prototypers

Requirement: ***[Multiple Prototypers]*** $\mathcal{PE}$ shall support multiple prototypers working on the same prototype.

As many programming projects involve multiple programmers, so the prototyping process will sometimes involve multiple prototypers or designers. Therefore, $\mathcal{PE}$ should support version and configuration management. The persistent object base should be designed to support multiple prototypers.

4. Related Research

Introduction

We now address some questions about relatively long-term research in related areas. This is not intended to be an exhaustive list of related research areas, nor is it intended to be a list of the only research areas we feel deserve attention with regard to prototyping.

Prototyping Languages as Production Languages

All prototypes are programs, but most are not suitable for production use, usually because they lack efficiency or robustness. Since programs and prototypes are distinguished by their intended use, not the notation in which they are expressed, one is tempted to hope that the same language could be used to construct both. In fact, today most prototypes are produced in languages designed for building production programs. On the other hand, the basic premise behind *PS* is that a special purpose prototyping language and system can greatly facilitate developing and analysing prototypes.

Other parts of this report have explained why conventional programming languages are not suitable for building prototypes. However, whether it will be possible to design *PL* such that it is suitable for building production programs as well as prototypes is still an open question. It is important that this not be a requirement of *PL*, but it would certainly be a desirable property.

There are several technical reasons *PL* could fail to be suitable for building production programs:

- *PL* might not be suitable for building programs that have to be maintained and modified over a long period of time.
- There could be functionality achievable in a particular programming language that is not achievable in *PL*.
- *PL* programs might not be robust enough for production use.
- The production program might have to run on a machine or in an environment that does not support *PL*.
- *PL* programs might not be acceptably efficient.

If *PS* were ill-suited to production programming because *PL* was not suited for building maintainable and modifiable prototypes, it would fail as a prototyping system. Though prototypes may be much smaller than the production program being prototyped, they will not always be small. Even if the prototypes are an order of magnitude smaller than the final program, they can still be large enough to raise all of the problems associated with building large programs. The key to rapid development of prototypes is likely to be reuse of part of other prototypes. This is only practical in an environment in which programs are relatively easy to maintain and modify.

In the long run, $\mathcal{PL}$ must provide the union of the functionality of the target languages. However, we expect that early versions of $\mathcal{PL}$ may do so only by allowing calls to them. For example, early versions of $\mathcal{PL}$ may not directly support distributed or other parallel programs.

Since $\mathcal{PL}$ will be used to explore large spaces of possible programs, it must facilitate making, installing, and testing changes. The flexibility implied by this may conflict with linguistic features designed to ensure program robustness. For example, it is widely accepted that requiring certain kinds of syntactic redundancy lead to more reliable programs. Yet requiring such redundancy may conflict with the goal of being able to make changes quickly.

Portability and efficiency of $\mathcal{PL}$ are closely related—they both depend upon advances in compiler and other implementation technology. If we could write highly portable systems and compilers, $\mathcal{PL}$ would be a very usable programming system as well as prototyping system. Unfortunately, experience indicates that neither of these is easy to do—and that the combination of portability and efficiency is even harder to achieve. Building a portable compiler is only the tip of the iceberg.

Portability and efficiency should not be requirements of initial $\mathcal{PL}$ implementations. To do so would place unacceptable constraints on the expressive power of $\mathcal{PL}$. However, research into the production of efficient portable implementations of powerful programming environments should be encouraged as part of the overall $\mathcal{PL}$ effort. Initially, this research can proceed independently of the design of $\mathcal{PL}$. However, as various $\mathcal{PL}$ features are proposed serious attention should be devoted to how they can be implemented. $\mathcal{PL}$ designers should be encouraged to design complete *languages (not merely kernels)* so that they can get feedback about how their language designs interact with implementation technology. Particular attention should be paid to the impact of various features on the runtime environment.

Transforming Prototypes

Transforming $\mathcal{PL}$ prototypes into production programs in the target language is a very hard problem. The root of the problem lies in the potential differences between a prototype and a production program. These differences have several sources as follows:

- The natural way to express something in $\mathcal{PL}$ may not have a natural translation into the target programming language. If one has to maintain the target language program text (rather than the $\mathcal{PL}$ text) ugly programs are not acceptable.
- One may want the global structure of the production program to quite different from that of the prototype. These differences may arise from efficiency considerations, from attempts to allow for expected modifications, or from robustness considerations that were not present in the prototype.
- The behavior of the production program will be different from that of the prototype. The purpose of a prototype is to provided feedback about the program, not to provide a complete

set of requirements. One may make changes because of things learned from the prototype. One may make changes in order to make particularly efficient implementations possible. One may make changes to handle cases ignored by the prototype, and this may make it desirable to change the way other things are done.

Despite the problems implied by these differences, this is an area of research that should not be ignored. While the probability of success may be low, the potential payoff is high.

Research on $\mathcal{PL}$ to $\mathcal{PL}$ transformations also offers substantial possibilities. There are four distinct kinds of transformations to explore. They differ in the relation between the original $\mathcal{PL}$ program, call it P, and the program into which it is transformed, call it Q.

- Behavior preserving: Client programs (i.e., programs that make use of P and Q) can detect no difference (including efficiency) between P and Q. These transformations are not likely to be terribly interesting.
- Interface preserving: Client programs can detect no difference between P and Q's behavior, except possibly with respect to efficiency. For programs in which timing differences can affect behavior these transformations are not behavior preserving.
- Correctness preserving: Client programs may be able to detect differences unrelated to efficiency between P and Q. However, Q meets P's specification.
- Non-preserving: Q may not meet Q's specification. At the lowest level of abstraction one can use a text editor to transform one program into another. However, one can imagine many higher level operations that could be used to transform one program to another in a more systematic way.

  One approach is to limit the domain of a program to increase efficiency. Examples of this approach include substituting floating point arithmetic for infinite precision arithmetic or limiting the length of identifiers. Another approach is to enlarge the domain or to generate a program that on the surface seems to do rather different things than the original. Examples of this approach include generalising a program that sorts integers to a general sorting routine, changing a parser for one language into a parser for another, and turning a Fortran compiler for Vax into one for a Sun.

  Understanding how one might program using such a transformations is a difficult but interesting research problem.

Specification and Prototypes

It is tempting to hope that given a suitable prototyping language one could avoid separate specifications by treating the prototype as the specification. We believe, however, that specifications and prototypes should be viewed as complementary.

An important, perhaps the main, purpose of a specification is to provide precise and easy-to-read module level documentation of interfaces. This documentation facilitates system design, integration, and maintenance, and encourages reuse of modules. Prototypes are less likely to be suitable for this purpose.

Research into the complementary roles of definitional specification and prototyping should be encouraged as part of the overall *PS* effort. This research should deal with specifications of pieces of programs written in conventional languages and pieces of prototypes written in *PL*. Specifying *PL* modules should be particularly valuable. If prototypes are to be developed rapidly, they must be built using previously written components. Given that one is relatively unconcerned about efficiency in a prototype, there is reason to hope that components will be more easily reusable than in production programming. The availability of good specifications can encourage reuse.

It is likely that developing a *PL* specification language will involve considerable research. A specification should provide all the information needed to write programs that use the specified component. A critical part of this interface is how the component can be combined with other components. Combining mechanisms differ from programming language to programming language, sometimes in subtle ways. Specifications written in a language tailored to a programming language are generally shorter than those written in a universal specification language. They are also clearer to programmers who implement components and to programmers who use them.

PL is likely to incorporate features not found in conventional languages. Furthermore, conventional features are likely to be combined in unconventional ways. Research will have to be done to devise languages and techniques well-suited to specifying *PL* modules.

Libraries of Modules

An important goal of *PS* is to encourage the reuse of program modules. This requires the existence of a collection of generally useful modules and some mechanism for combining them. However, that is not enough. By way of analogy, consider a collection of books. The utility of the collection depends not only on its contents but also upon the way they are organised (e.g., the Library of Congress System) and the tools available for discovering what books are contained in the collection (e.g., a card catalog).

At present no semantics-based system for organising program modules exists. Almost all programmers use some *ad hoc* hierarchy, sometimes supported by a browser. This organisation is rarely useful to others. Research into general techniques for categorising modules seems difficult but useful.

There is also no widely accepted tool comparable to a card catalog. Hierarchical browsers tend to lose their effectiveness when the number of modules grows large. One might be able to build a useful keyword based system on top of a conventional database. It would be more interesting, however, to investigate systems that performed retrievals based directly on either the bodies or

specifications of modules.

Programming Distributed and Parallel Systems

It is quite clear that distributed and parallel programs will grow in importance in the future. Systems of the future may be networks of predominantly parallel computers.

Work in this area should emphasize the construction of parallel programs and systems. At this point in time, the research community clearly needs experience and data.

Researchers should be encouraged to develop new parallel programming languages, to build high quality implementations of these languages, and to implement prototype systems in them. The exploration of the relationship between modularity (e.g., through abstraction) and efficiency seems particularly important. One aspect of this is understanding to what extent one can abstract from underlying computer architectures.

The development of tools to support parallel programming languages is also important. Debugging and performance monitoring and tuning of parallel and distributed programs is extremely difficult and not well-understood.

Programming Languages

It is not easy to distinguish prototyping languages from programming languages—it is mostly a matter of emphasis. These differences in emphasis, however, can lead to substantial differences as one makes the tradeoffs involved in designing languages and environments.

There is obvious utility in continued research in which the tradeoffs are made in favor of the programming language biases. Specifically, research should continue into designing languages in which high degrees of efficiency can be achieved.

Programming Environments

Many of the tools found in conventional programming environments will play prominent roles in *PS*. These tools include testing tools, debuggers, and performance analyzers. *PS* will likely provide an excellent context in which to push the state of the art in these tests. There is *a priori* more emphasis on the environment in *PS* than is normally the case. Also, designing the environment and the language in concert should yield considerable synergy.

In some cases, however, it may be easier to push the state of the art of the tools in a conventional context. The research community's extensive experience constructing and maintaining programs written in conventional languages has helped to clarify major problems with current tools. Furthermore, there is little doubt that languages such as Ada and Common Lisp will be with us for many years. Therefore it pays to make substantial investments in improving their environments. Finally, because there will be fewer variables, it may be easier to push the state of the art on particular tools in the more stable environment associated with conventional programming languages.

Current programming environments are not as well-integrated as they should be. For example, the various debugging, performance monitoring, and static analysis tools do not present their results in one display. It is not possible to graphically compose simple programs. Informal data such as notes from a prototyper to himself and his colleagues are handled by computer mail, files, and paper, and not by the environment.

Current programming environments require the programmer to manually manage the look of his screen and to spend a lot of time switching among tools. There is much computer science and human factors research whose results could streamline the prototyping process.

Part of the prototyping environment is a persistent object base, which is either implemented in terms of a database or shares many of the same problems with databases. In order to smooth the use of a persistent object base, research must be done to solve some of the following example problems. It should be possible to store the types of objects required by a prototyping environment, which might include running programs and the environment itself. It should be possible to retrieve these objects quickly enough in a multi-user context. In multi-user situations it might be necessary to be able to compose possibly mismatched object bases.

Prototyping versus Programming

In the best of worlds, there would be no difference between the environments used for producing prototypes and those used for producing production programs. In such an environment producing a production program would often involve incremental evolution of a series of prototypes. Of course, that is exactly the way most research software gets built today. Languages that provide good support for modularity and abstraction lend themselves to that paradigm. A problem is that the designers and implementors of these languages and systems have usually emphasised features aimed at producing production software, for example, efficiency, rather than those aimed at producing prototypes.

The main body of this report emphasises the development of languages and environments aimed specifically at prototyping. If that is successful, people will surely work on enhancing *PL*'s and *PS*'s suitability for producing production programs. Another approach is to start with an existing programming language and system and work on enhancing its suitability for producing prototypes.

Operating Systems

One fear we had while thinking about $\mathcal{PS}$, especially $\mathcal{PE}$, was that existing operating systems were so closed and lacking enough features that it would be difficult to provide the sorts of debugging and monitoring information required for a useful environment. Similarly, many performance problems we anticipate with $\mathcal{PL}$ programs could be alleviated with sufficient control over the lower levels of the operating system.

We do not make any remarks regarding whether that operating system work be undertaken as part of the core $\mathcal{PS}$ work, but we would not be surprised to see design groups propose operating system (and possibly hardware) work as part of their work plan. And certainly we would like to see a resurgence in operating system research.

The same remarks hold for computer architectures.

Annotations and Functors

Harlan B. Sexton
Stanford University

1. Abstract

Annotations are a generalization of hypertext defined for use in a prototyping architecture. They have proved to be surprisingly adaptable and general. This note is an attempt to provide a formalism that explains part of why this is so.

2. Introduction

Annotations are the key to integration of the components of a prototyping environment. Informally, annotations are links between objects; i.e., generalized *hypertext*[1] or the building blocks of arbitrary *relations*.[2] Obviously very little can be said about something as general as a link, but the use of annotations in the prototyping system has proved surprisingly useful. In other words, a construct that is, apparently, so general that it can have no structure of its own has proved to be very useful in the implementation of a complex programming environment. This means, obviously, that there is, in fact, structure somewhere, and the purpose of this note is to elucidate this structure.

To motivate our formal model of annotations, we give an outline of the relevant aspects of the prototyping environment, and the way that annotations are used there. This description is by no means a complete one; the interested reader is should consult [Gabriel 1990], from which most of this outline was taken.

2.1 *Prototyping Environment Architecure*

An *environment* is a set of tools integrated harmoniously for the purpose of accomplishing some task or set of tasks—in particular, this set of tools must be able to work well together as judged by a user of the environment. Often, this is simply being able to share information, but it certainly requires being able to apply tools at the right time and to be able to retain information.

A programming environment is a set of tools designed to facilitate the tasks associated with developing software. These tools include, but are not limited to, editors, compilers, debuggers, and linkers. Even more important than the tools provided is the mechanism by which these tools are integrated. This mechanism provides a well-defined, flexible, and extensible architecture for the components of the environment.[3]

The work was supported by DARPA, contract number N00039-84-C-0211.

[1] *Hypertext* is a commonly used term that has many different definitions, but in general it is used to refer to mechanisms that link related pieces of text in a more or less active way.

[2] By a *relation* here we mean a mathematical one—that is, a set of n-tuples.

[3] The description of this architecture below makes free use of all of the terms defined in the appendix.

Clearly, the requirements of such an architecture are driven by both the needs and expectations of those using the system and those extending it. Specifically, for the user a programming environment must provide tight integration of tools and the information they provide. One way to accomplish this is with a closed implementation of the environment in which the programming tools are designed together to work on the same representation,[4] but this fails those who would augment the system. (Also, it generally proves to be more difficult to make such tightly coupled environments work well on distributed computer systems.)

In our environment tools are clients served by a kernel that effects the integration. Tools and the kernel do not operate on the same representation. Instead they operate on the same abstract interface to shared information. In other words, the kernel handles keeping track of the information derived from the tools and relationships among such information, and the (client) tools provide the information and the relationships.

2.2 *Protocols*

The architecture of the environment is built on a set of ***protocols*** that are used to pass information among the different components of the architecture. For example, the kernel learns about the structure and significant contents of a particular program source text by passing that source to the compiler which returns such information as where definitions are made and where those definitions are referenced.

The use of protocols this way enables knowledge appropriate to a particular tool to remain local to that tool and for the implementation of activities appropriate to shared information extracted from that tool to be similarly local to the kernel or other clients. For example, in a programming environment the kernel has no detailed knowledge about any particular programming language, but it has knowledge about programming and the general concepts of programming languages. This knowledge is built into a *tool protocol server* which attaches behavior to objects and their annotations. For example, information about a program in the form of objects that represent the program text and annotations such as cross references and documentation would be generated by compiler server and retained by the kernel.

2.3 *Types*

As we said before, the kernel is the site of the integration of information and maintenance of relationships among different tools. In particular, the kernel "knows about" different tools via a *type system* that is in the extensible part of the kernel. The addition of a new tool to the system is accomplished by adding the appropriate type, *e.g.*, parse

[4] For example, in such a system a "procedure" might consist of structure containing source code, a parse tree for the code, a "symbol" table, etc. Then, when the editor made changes to the source code part, the compiler could make changes to the parse tree and update the symbol table so that the association between substrings of the source and nodes in the tree would be correct. In such a system the various tools usually end up tightly intertwined.

trees, to the kernel's type system,[5] and providing the code that connects relevant pairs of types to their associated tool, *e.g.*, text and parse trees to a parser. In other words, the protocol is a way of "extending" the mapping done by the tool (taking text and transforming it into a parse tree) into a mapping between kernel types (associating substrings in the source code with subtrees in the parse tree).

2.4 *Annotations*

An annotation is an object along with an associated set or sequence of other objects. An annotation acts as a link among those other objects, and is an instance of a class to which methods can be attached. An example annotation is the simple link which is associated with a sequence of two objects. The first of the two objects is the *source* and the other is the *destination* of the link.

More formally, an annotation is a pair (A, S), where A is an instance of a class that is a subclass of the class $\mathcal{A}$, which represents annotations, and S is either a set $\{O_1, \ldots, O_n\}$ or a sequence $\langle O_1, \ldots, O_n \rangle$, where each O_i is an instance of some class. Further, there exists an $F: \mathcal{A} \rightarrow \mathcal{S}$, where $\mathcal{S}$ is the set of all subsets and subsequences of objects, such that if (A, S) is an annotation, $F(A) = S$. Therefore, we can unambiguously denote the annotation (A, S) by A.

Furthermore, there is a function f such that given any object o, $f(o) = \{(A, S) \mid o \in S\}$. That is, given any object it is possible to find all annotations that involve it.

The class of an annotation provides a set of operations that can be performed on instances of that class. The operations on an annotation are carried out by the kernel. The results of carrying out an operation often result in the presentation client presenting new or altered material to the user.

2.5 *Example*

Let us illustrate these definitions by an example of the use of annotations in the implementation of a user interface.

Consider a client process displaying information obtained from a second client (such as a compiler). The presentation client knows how to display graphical and textual information along with a visual display of annotations. When the user requests the available operations for an annotation, the presentation client requests the kernel to provide a selection structure. This selection structure is presented to the user, and once a choice is made, the kernel is queried about how to proceed. This is an example of a style of user interaction—the presentation client might present annotations differently or it might invoke the protocols differently for better or more appropriate responsiveness to the user. However, the partition of knowledge is constant over all pairs of the presentation client

[5] Adding a new type entails defining various methods (or protocols) to do conversion of objects from one type to another—these methods define the lattice structure for the type system. This will be discussed further below.

and kernel. In concrete terms, if two objects are linked by an annotation, the presentation client will be able to offer the user a means to get from source to destination and destination from source.

Note that the presentation client accomplishes such a user interaction given only the information that a particular object has a particular annotation attached to it. The presentation client does not know anything else about the annotation. The presentation client has a built-in means of interacting with annotations.

In general, an annotation connects objects, so to make use of them it is sometimes necessary to "invent" objects to be connected. For example, making connections between source code and the structural elements of the program this code represents is clearly desirable. More generally, a "fundamental" presentation for most programming object is textual, and so we naturally wish to create annotations between the "actual object" and its textual representation. In the case of pure text, an artificial object called an *extent* is created to represent an arbitrary piece of text that is to be annotated. An extent is a contiguous region of text to which annotations can be attached. Extents are allowed to arbitrarily overlap, and several annotations can be attached to the same extent. The system is made to "understand" *extents* by adding them to the kernel's extensible type system (as a super-type of type text).

2.6 *Annotations are Active*

There *is one other* aspect of annotations that needs to be emphasized here. An annotation is *active* in the sense that methods may be attached to annotations by being associated with a class. When a tool, such as a compiler, is being sent source text, any annotation encountered during the process may have an associated method invoked to determine the text to be sent in its place. For example, region locking can be implemented with annotation methods. When locked text is sent to the presentation client, the method associated with the annotation causes the correct commands to be sent to the presentation client to cause the text to be locked in the editor.[6]

Typically, methods are run in the kernel, possibly in the tool protocol servers, but protocol server methods can also invoke methods run in clients.

3. Annotated Categories

The model presented here is an attempt to formalize the manner in which the architecture manages and integrates structural information from different program development tools. The model treats annotations as the building blocks of "enhancements" to certain types of functors acting between categories that embody different ways of representing such entities as programs. The use of category theory here may seem unnecessary (and it certainly isn't essential), but categories are generally used to represent different kinds of models for axiomatic entities, and so it is certainly evocative to use this language. However, all

[6] Obviously, such functionality requires that the presentation client and other tools preserve the semantics associated with region locking.

the categories under consideration here are small,[7] and the prerequisites for understanding the presentation here are simply knowing the basic definitions. Mathematically oriented readers can consult [MacLane 1972], or almost any introductory graduate-level abstract algebra text. Readers desiring a computer science oriented text can consult numerous texts, too, such as [Arbib 1986].

3.1 *Basic Model*

The basic idea of our formal model is the following.

A programming tool is a functor that relates complete objects of one category with complete objects of another (or the same) category; in other words, the defined mapping is monolithic with respect to objects. However, the objects in "programming categories" have additional important structure, and the mappings performed by these programming tools do, in fact, use and preserve this structure. Further, the details of how this structure is used and preserved are important, and those details are annotations.

Let us make this specific with a simple example. A parser is a map from source code to something like expression trees (depending on the parser). The source code has structure given it by the grammar for the language, of course, but more importantly and more naturally it has structure because source code is a subtype of text, and so it inherits structure from that type. Similarly, expression trees have structure, either their own intrinsic structure or that inherited from some supertype such as trees. In any event, the mapping from a program to its tree is only defined on those terms—give me a program and I'll give you this object back. The fact is that there is a lot of information sitting around inside the parser when it is finished that, if mapped back to the source code, would be very useful; information about binding scopes, variable declarations and uses, function calls, etc. Annotations are the means by which this mapping is handled, and they are a "natural" extension of the parser, a map from source code to parse trees, to a map from text to (potentially) some supertype of parse trees.

The goal of this section is to define all of the terms needed to express this in a formal way. In the next section we will apply these definitions to a couple of examples, and discuss more of the connection between the model and the environment.

3.1.1 *Types*

In defining the terms of our basic model we shall use the idea of a *type system*. In the usual case, a type system $\mathcal{T}$ is a lattice of subsets of a given set X, say. In this instance, for $t \in \mathcal{T}$, we would say that $x \in X$ is of type t if and only if $x \in t$. In the most general case, $\mathcal{T}$ is still a lattice and the t are still sets, but now there is an associated inclusion map from t into u for every pair of types such that $t \leq u$, and further these maps commute

[7] Their underlying classes are all sets, and the morphisms are generally trivial, as well. These categories are of the type usually referred to as *Kiddie-gories* by sophisticated users of this dismal subject.

in the obvious way.[8] A type system may (and often does) have other coercion methods than those defined by the inclusions. (Such coercion methods may be partial functions, but they generally do have commutativity requirements.)

3.1.2 *Structure*

Most of the types that we shall consider will also be what we refer to here as *well-structured.* Informally, we say that an object has structure if it can be "decomposed" into "simpler" objects. For the purposes of our discussion we shall define a decomposition of an object O to be a set of elements $o_1, ..., o_n$ belonging to the type of O. More specifically, given a type t, define the decomposition set for t, denoted DS_t, as the set of all finite sets of finite, non-empty subsets of t. The idea is that, given elements $x \in t$, and $D \in DS_t$, we can think of D as being the collection of all the different decompositions of x. For example, if t is the integers, then x might be 210, and D might be $\{\{2, 105\}, \{6, 35\}, \{6, 5, 7\}\}$. We refer to the elements of D, such as $\{2, 105\}$ as decompositions of x. Notice that, by the definition, D could be the empty set. (It isn't obvious that we have to have the set D be finite, but there seems absolutely no reason to admit the possibility of an element having an infinite number of possible decompositions, so we go ahead and require it.)

A *decomposition structure* for t is defined to be a function $d: t \rightarrow DS_t$. We say that $x \in t$ is *atomic* (with respect to d) if and only if $d(x)$ is null. Notice that a decomposition structure can be used to define a directed graph structure on a type t by connecting each element of $x \in t$ to all elements in any decomposition of x. A node in this directed graph is a leaf if and only if it is atomic.

> **Definition:** The type t is **recursively-structured** (with respect to d) if and only if the directed graph induced by d is acyclic. The type t is **well-structured** if and only if it is recursively-structured and every path in the directed graph is of finite length.

It is also important that the decomposition structures for the individual types in a type system are coherent. A *decomposition structures map* for a type $\mathcal{T}$ is a function D that maps each $t \in \mathcal{T}$ to a decomposition structure d. We say that D is *coherent* if and only if it commutes with the inclusion maps; that it, if t is included in u, then every decomposition of an element in t under $S(t)$ is also a decomposition of that "same" element in u under $S(u)$.

> **Definition:** Let $\mathcal{T}$ be a type system and $\mathcal{D}$ be a coherent decomposition structures map. A type system $\mathcal{T}$ is **structured** with respect to $\mathcal{D}$ if each type t is recursively-structured under $\mathcal{S}(t)$. $\mathcal{T}$ is, similarly, **well-structured** under $\mathcal{D}$ if each t is under $\mathcal{S}(t)$.

It is obvious that with reasonable definitions types such as strings and trees can be regarded as well-structured. Note that in the extreme case we can suppose that every

[8] In practice these more general type systems may not "actually" have top and bottom elements, but we can always make the bottom be the null set and the top be the direct limit of the sets and inclusions, so we can assume these are present if we want to.

object in the type is atomic, but we should avoid this case when we can. Obviously we can't be totally arbitrary how we fit structured typed in the type hierarchy since we must make sure that the structural decompositions are preserved, too, but this happens automatically most of the time.

3.1.3 *Typed Categories*

> **Definition: A system of typed categories** is a pair $(\mathcal{T}, \mathcal{S})$, where $\mathcal{T}$ is a type system and $\mathcal{S}$ is a set of categories such that for each $\mathcal{C} \in \mathcal{S}$, there exists $t \in \mathcal{T}$ such that the underlying class of objects of $\mathcal{C}$ is a subset of t.

In other words, a system of typed categories is a type system and a set of categories such that any elements of a category is of a specified type. The systems with which we use annotations generally involve objects that are structured in some non-trivial way, so we will normally only care about such systems.

> **Definition: A structured category** is a typed category such that the type is well-structured. A system of structured categories is defined in the obvious way.

3.1.4 *Annotated Functors*

Consider the parser again. The parser defines a map from the category of source code to that of parse trees, and since we don't have any non-trivial morphisms that makes it a functor. However, just saying that it is a functor isn't very useful, since any map from source code to trees is one, and the parser has a great deal more structure than that. In particular, subtrees of the parse tree correspond to substrings of the source code. In general, we can formalize this for structured categories by saying that a functor is compatible with a "destructuring" (that is, a specific decomposition).

Now let us define a "structural" annotation of a functor.

Let $\mathcal{F}: \mathcal{C} \rightarrow \mathcal{D}$ be a functor, where $\mathcal{C}$ and $\mathcal{D}$ are structured categories. Let $t_{\mathcal{C}}$ and $t_{\mathcal{D}}$ be the types of $\mathcal{C}$ and $\mathcal{D}$, respectively. Finally, let the objects D_i be in the range of $\mathcal{F}$, and let d_i^j be decompositions of these objects. Then we say that $\mathcal{F}$ is compatible with the d_i^j if and only if there exists a partial function f from $t_{\mathcal{C}}$ to $t_{\mathcal{D}}$ that extends $\mathcal{F}$ and such that for every C_i in $\mathcal{F}^{-1}(D_i)$ there exist decompositions c_i^j of C_i such that $d_i^j = f(c_i^j)$. We say that the function f is a structural annotation of $\mathcal{F}$ with respect to these objects and decompositions. We say that f is a full structural annotation for $\mathcal{F}$ if it is an structural annotation for every object and decomposition in the range of $\mathcal{F}$.

> **Definition: An annotatable functor** is a functor between structured categories that has a full structural annotation.

In other words, an annotatable functor is one for which all structural information of target elements can be "seen" in source elements.

3.1.5 *Remarks on the Basic Model*

As we said above, the content of this model is (or at least is supposed to be) that the functors that correspond to most programming tools are annotatable, and these functors are generally one of two kinds. The first kind is a functor whose domain category is in some sense free, or at least much freer than the range category. The second kind of functor (which may also be of the "first kind") is one that decomposes the argument while "computing" its value, and the value cannot be decomposed in any significantly different way.

Two rules of thumb are that functors from text categories will frequently be of the first kind, and many functors that are algorithmic are of the second kind. (In the following, let $\mathcal{F}: \mathcal{S} \rightarrow \mathcal{X}$ be a functor.) The reason for the first rule, when the objects of S are strings, is that any sub-object of X will almost always correspond to a substring of something in S—that is, any structure that is "impressed on" text usually doesn't split single characters. The reason for the second rule is that if $\mathcal{F}$ is algorithmic, the algorithms often uses some sort of recursive decomposition to get from s to x, and such methods tend to preserve decompositions.[9] It is always possible, for any functor, to make an analogous one which is annotated, by defining a new range type with fewer decompositions,[10] which has the effect of limiting the amount of additional structure enough to make the functor annotated.

4. Module Description System

The purpose of this section is to outline a system for defining and enforcing structural decompositions of programs. We begin by first giving an overview of the abstract model we shall use for describing program modularization. Later we make precise some undefined terms used in the model. Before we start, we need a bit of notation. Let us denote by $\mathcal{S}_{\mathcal{L}}$ the category of programs for the language $\mathcal{L}$. Also, unless we state otherwise, in this section by *graph* (directed or not) we mean a graph such that no edge connects a node to itself.

4.1 *Overview*

Associated with the category $\mathcal{S}_{\mathcal{L}}$, there exists a set $\mathcal{A}_{\mathcal{L}}$ (generally infinite) of objects which we refer to as the set of *code atoms* for $\mathcal{L}$. Each object in $\mathcal{A}_{\mathcal{L}}$ has a unique associated string which is referred to as the atom's textual representation. In other words, the language $\mathcal{L}$ has a well-defined set of program fragments that are the smallest units of text under consideration here. As a rule of thumb, a code atom is a piece of code that is a function or variable definition.

A program P in $\mathcal{S}_{\mathcal{L}}$ has a directed acyclic graph structure $\mathcal{A}(P)$ associated with it such that the leaves of the graph are elements of $\mathcal{A}_{\mathcal{L}}$, and the program text of P has a unique

[9] A notable exception to these rules is a compiler which uses global optimization techniques. In this case, it may easily happen that there are no structural similarities between source and compiler output, only functional similarity.

[10] This is precisely what one would do in the case of an optimizing compiler.

substring which is equal to the text of each "leaf" atom. In other words, the program P is composed of the text of code atoms plus whatever "glue" is needed to hold them together. Note that a given atom might occur in more than one place in a program.[11]

Also associated with a program P is a directed graph, $\mathcal{D}(P)$, whose nodes are the leaves of $\mathcal{A}(P)$ and whose edges are the "dependencies" between these nodes. Informally, the presence of the edge from n to m in $\mathcal{D}(P)$ means that n defines something and m uses it, but we shall define this more carefully below. Again note that the code atom associated with a leaf of $\mathcal{A}(P)$, and therefore of a node of $\mathcal{D}(P)$, may not be unique. For example, in C static variables with precisely the same code as definitions can appear in multiple files.

A *modularization* of a program P is a sequence of graphs $\Gamma_1, ..., \Gamma_N$, with $\Gamma_1 = \mathcal{D}(P)$, and such that each Γ_{i+1} is a quotient graph of Γ_i. We denote by $\mathcal{M}(P)$ the set of such modularizations, and for a particular element of $\mathcal{M}(P)$, the nodes of Γ_N are referred to as the "modules" of P. The essential idea behind a module system is to equip a program P with a modularization M of "minimal complexity" (for some measure of complexity). We shall have more to say about this later, too.

4.2 *Further Details*

There are several points that need clarification. These are the nature of $\mathcal{A}_{\mathcal{L}}$, how to get from P to $\mathcal{A}(P)$ (and hence the nodes of $\mathcal{D}(P)$), and how to determine the edges of $\mathcal{D}(P)$.

The set $\mathcal{A}_{\mathcal{L}}$ is axiomatic—that means, in practice, that it is defined by the implementation of the module system for a specific language. Thus, it is assumed also that this knows how to "recognize" these atoms within a program, and so in effect $\mathcal{A}(P)$ is axiomatic, too. Typically each atom is a code fragment that "defines" a "name", and the compiler protocols recognize the extent of the definition *and* the references within the definition to other names. Thus, the compiler protocol can construct $\mathcal{D}(P)$ for the module system.

The fact that the compiler determines the links that determine the modules has the implication that the module system needs to account for the "sequential" processors of a program, including any macro expanders, compilers, and linkers that lead to the creation of links. In particular, the module system should be able to "make" the run time version of a program from a modularization.

In a sense, the purpose of such programming practices as data abstraction is to *create* the links in $\mathcal{D}(P)$. If the abstraction is effected by methods, then the links are all run time in nature and are handled naturally by the compiler. However, in some cases the abstraction is the result of a macro, and in this case the dependency is compile time. This complication is significant, as such compile time dependencies can introduce further

[11] The initial stage of the construction of the "atom graph" would result in a sort of high-level parse tree. The reason the graph may actually be a DAG instead of a tree is that, in some cases, there are code atoms that define precisely the same thing that appear in more than one location in the source.

run time dependencies, and if the macros are sufficiently powerful, then the nature of these run time dependencies cannot be determined until compile time. As a result, it will be necessary for a compile time module to specify which runt time dependencies it can introduce to allow static processing of the module description. In other words, it is possible to determine the dependencies of "macro-free" code by inspection, but if the code depends on the results of a reasonably powerful preprocessor, then declarations will be needed to allow a rigorous analysis of a proposed modularization to be done.[12]

This static analysis of a modularization is an essential feature of any module system. The idea is that a module system should be a design tool, not just a bookkeeping tool. Good modularization happens only as the result of the design of a program, not after the fact.

5. Appendix: Various Definitions

To describe the architecture we make use of the following terms:

Server: A program that provides service or functionality to a variety of other programs called *clients* through a message-passing medium such as byte streams in Unix. A server rarely performs user interface activities.
Client: A program that communicates with a *server* for the purpose of information exchange. A client might engage the user in a dialogue. Though the terms *client* and *server* are arbitrary, a client generally communicates with relatively few other programs while a server generally communicates with many other programs.
Kernel: A server that performs the fundamental actions in an environment.
Protocol: A language designed to pass data and control between a client and a server. Typically a protocol is a series of messages and containers of information encoded in a common medium, such as byte streams or ascii text.

References

[**Gabriel 1982**] Gabriel, R. P. et al *Common Prototyping System Final Report*, to appear.

[**Arbib 1986**] Manes, E. G. and Arbib, M. A., Algebraic Approaches to Program Semantics, Springer-Verlag, 1986.

[**MacLane 1972**] Mac Lane, S. Categories for the Working Mathematician, Springer-Verlag, 1972.

[12] Notice that these declarations can be tested, if not verified, at compile time.

Using CLOS-like Concepts in a Prototyping System

Richard P. Gabriel
Stanford University

We have been working for about 9 months on the design of a prototyping system, which is a language and an environment for creating mixed Lisp and Ada prototypes. The environment and language use CLOS-like concepts. This precis describes the language concepts we are developing.

Traditional languages like ADA use a model of sequential control with a side-effectable memory. Combination or aggregation mechanisms are used to compose complex data structures from simpler ones. Most programming languages provide extensive mechanisms for defining abstract data types from primitive ones, but very few programming languages provide mechanisms for defining control abstractions (besides procedures) or other non-data abstractions. We believe that prototyping is largely a process of composing or aggregating complex prototypes from simple ones, and sometimes such composition is performed with source code and sometimes not. Therefore, a prototyping environment is one whose primary function is to compose and control existing source and executable mixed language programs and modules.

In particular, we are interested in developing a richer computational model for a prototyping language than the ones used by Ada and Lisp so that prototypes can be manipulated by manipulating this model.

Over the last 6 months we have adopted the following approach:

1. Select a problem for which we wish to develop a prototype.
2. Suppose some existing code that accomplishes some part of the task or a similar task.
3. Determine the program that should constitute the prototype.
4. Infer the partial programming model from that prototype.

The following constraints are also being observed. Note that several are environmental rather than programming concerns:

1. Do not require that the existing code be copied out of an existing configuration. That is, the environment should take care of versions and merging conflicts, up to a point.
2. Where possible, enable the prototyper to alter the context of execution (or compilation, if that's the way you want to think of it—they are the same thing) in order to achieve the behavior needed for the prototype.
3. Use *annotations* to enable the prototyper to keep a record of transformation made by hand or by machine and the reason for the transformation so that the history of

The work was supported by DARPA, contract number N00039-84-C-0211.

development is retained. A piece of source code can be annotated by the program that will transform that source to the desired form. The *abbreviation* of the annotation will be the transformed code. The original source code remains intact. When compiling the source code, the compiler will fork off a process to either perform the transformation or retrieve it, and supply the character stream (or tokenized stream) in place of the original text. Annotations and abbreviations will be described in the next section.

4. Do not rule out visual presentation or language, but insist on a textual representation for every prototype.

1. Annotations and Abbreviations

An ***annotation*** is a generalization of hypertext. Annotations are used to integrate the various prototyping tools and to provide an extensible base. An annotation is a relation that links two or more objects in the environment. An annotation is an instance of an extensible class hierarchy. Annotations are active in the sense that methods can be associated with them that are triggered whenever a tool of a certain class operates on an annotation of a certain class that links objects of certain classes. Normally a region of source code is annotated with some information.

The second part of the annotation model is ***abbreviations***. When an object is annotated with some other things, the annotation in general must be displayed when the object is displayed. There are several options. First, the object can be displayed as usual but in a mouse-sensitive manner, so that the annotation can be displayed by interacting with the mouse-sensitive object. Second, the annotation can be displayed as an icon either in place of the object or adjacent to it. Third, the annotation and the object can be displayed in an abbreviated manner. For example, if a comment in a program is turned into an annotation, that comment could be displayed as follows:

```
(defun f (n)
  ;;; Logarithmic Fibonacci function....
  (...))
```

Here the one line comment is an abbreviation of a much larger comment that explains the algorithm. The derivation of the abbreviation could be by taking the first three words of the comment, or it could be some programmer-defined object.

Annotations and abbreviations form the basis for communication, integration, and user interface in the prototyping environment. For example, annotations are used in source level debugging: When a program is halted, the source code is displayed with each variable annotated with its current value. If a function is recursive, further annotations on each variable will provide the values it has in enclosing invocations, so that they may be simultaneously viewed in place in the source.

However, describing the environmental model for prototyping is not a goal of this precis.

2. Samefringe

In following the approach outlined above, one problem in particular has occupied us: The problem is samefringe. This problem was suggested many years ago as the simplest problem requiring multiprocessing to solve satisfactorily. As it turns out, it can be solved quite well and elegantly using serial techniques. The problem is as follows. Assume that T1 and T2 are two binary trees whose leaves are non-NIL atoms. Let L1 be the list of leaves encountered in a preorder tree walk of T1, and let L2 be the corresponding list for T2. Then T1 and T2 have the same fringe if L1=L2. In Lisp, the trees have the same fringe if, when printed, the same atoms appear in the same order. We have restricted leaves to be non-NIL atoms to simplify the problem without loss of generality.

One obvious way to think about how to program this is to envision a tree traversal program like this (in Lisp):

```
(defun traverse (tree)
  (cond ((atom tree) (report tree))
        (t (traverse (car tree))
           (traverse (cdr tree)))))
```

Here REPORT simply notes its argument somehow. First we want to set up two processes, one to traverse T1 and the other to traverse T2. Then we set up a third process to look at the atoms coming from each traversal. If any two corresponding atoms are not the same, the processes are terminated and the answer is nil. Otherwise, if both traversal processes end normally, the two trees have the same fringe if both processes terminated after reporting the same number of atoms.

We can assume that the traversal routine is given—it is part of the stock of programs available to the prototyper for re-use. The following comparison routine checks a pair of atoms from each tree:

```
(defun compare (x y)
  (unless (eq x y) (return-from samefringe nil)))
```

We can assume this is given as well.

Later when the visual approach to prototyping is presented we will see that the concept of "process" is possibly unnecessary to solve the problem. This is because the prototyping environment provides a mechanism to make two copies of the source code and to indicate that each will receive one of the trees to work on. Of course, the textual representation of the program will involve processes, and we will be solely concerned with that textual representation except in the section "Illuminated Code."

Given the basic approach outlined earlier, the following seems to be the naive code to solve the problem:

```
(defun samefringe (tree1 tree2)
  (qflet t ((compare (a1 a2)
              (unless (eq a1 a2)
                (return-from samefringe nil))))
    (labels ((traverse (tree)
               (cond ((atom tree) <report-tree-to-compare>)
                     (t (traverse (car tree))
                        (traverse (cdr tree))))))
      (qprogn t
        (traverse tree1)
        (traverse tree2))
      <test-for-same-number-of-atoms-reported-from-each-process>)))
```

The QFLET expression creates a process running COMPARE, and the QPROGN expression spawns two processes, each running TRAVERSE. The T in the QFLET expresssion indicates that a process is always to be created.

Now we must complete the program. The remaining presentation follows the lines of exploration we followed while trying to do that. Often studying such lines can be as illustrative as any of the intermediate points and certainly more than the endpoint alone. At the end of this precis we have the most recently developed version of the samefringe program, and it is not yet entirely satisfactory.

The first step is to solve the communication problem between the expression `<report-tree-to-compare>` and COMPARE.

3. Partially, Multiply Invoked Functions

The basic idea of partially, multiply invoked functions (PMI Fun ons) is to separate the process of coordinating the arrival of arguments from the process of executing the function on those arguments.

All calls to a function go through an interface to the actual implementation. The implementation of the function receives arguments by position, while the interface accepts only named arguments, provides for all defaulting, and coordinates the arrival of arguments for the function from multiple sources.

The Qlisp work produced the basic idea of PMI functions in late 1987, but the Qlisp formulation has a serious drawback. Namely, arguments need to have names, which often requires those names to be passed about as additional arguments so that the proper values were assigned to the proper parameter names.

Here is a simple example of the Qlisp-style technique:

```
(pmi-defun add-up (x y) (:summand1 :summand2) (+ x y))
```

This function adds up a pair of arguments, called X and Y. The expression

```
(add-up :summand1 1 :summand2 2)
```

simply produces the answer 3. However, one can partially invoke the function as follows:

```
(add-up :summand2 2) -> future
```

In this case the interface remembers the supplied argument and returns a future. A second call will complete the invocation and supply a value to the future:

```
(add-up :summand1 1) -> 3 = <the same future as above>
```

This technique, then, is similar to currying functions, but because all arguments to the interface are named, one does not need to curry in any particular order. One could term it ***dynamic currying***. All calls to a PMI function that supply arguments to the same invocation receive the same future as their value. A future is returned whenever some required arguments to the function have not been supplied to the interface by a function call. If a particular invocation of a function has returned a future, the value returned when all required arguments have been supplied is a realized future. This is to preserve EQ-ness of all values returned for a particular invocation.

Also, when not all arguments are supplied for a particular call, futures are supplied for unsupplied arguments. This way, PMI functions can be truly partially invoked.

For example, we can produce a list of the sums from two streams supplied by two processes as follows:

```
(let ((answer (make-queue)))
  (pmi-flet ((add-stream (x y) (:summand1 :summand2)
               (add-queue (+ x y) answer)))
    (qprogn t
      (loop ...
        (add-stream :summand1 <computation>)...)
      (loop ...
        (add-stream :summand2 <computation>)...))
    answer))
```

The PMI-FLET expression creates a local PMI function. The details of queue management are elided.

It is possible to use streams, pipes, or channels to write SAMEFRINGE and many other programs. It would seem that these other mechanisms are superior because they are simpler, more easily understood, and more intuitive. On the other hand, using these mechanisms requires "wiring up" networks of channels, either statically or dynamically. In either case there would be code to effect the wiring. With PMI functions the items being passed are sent to their destinations, labeled with the argument to which it corresponds, and there is no need to identify, centralize, or explicitly name sources of arguments, only

destinations. Thus it is easier to code up loose collections of processes that communicate variously with one or several central processes.

In this way PMI functions mimic pure message-passing, where the sender does not need to be connected to the receiver. The advantage of PMI functions over classical message-passing is that the recipient can be a function of many arguments, whereas the suppliers of those arguments need not be otherwise co-ordinated to send them.

Furthermore, PMI functions preserve functional style, and there is no restriction that a single invocation of a PMI function must pass exactly one argument. For example, suppose ADD-STREAM takes three arguments, sums them, and adds them to the queue. And suppose that each of the two processes supplying the summands alternate supplying one and two of the summands. We could write this as follows:

```
(let ((answer (make-queue)))
  (pmi-flet ((add-stream (x y z) (:summand1 :summand2 :summand3)
               (add-queue (+ x y z) answer)))
    (qprogn t
      (loop ...
        (add-stream :summand1 <computation>)
        ...
        (add-stream :summand1 <computation>
                    :summand2 <computation>)
        ...)
      (loop ...
        (add-stream :summand2 <computation>
                    :summand3 <computation>)
        ...
        (add-stream :summand3 <computation>)
        ...))
    answer))
```

This is possible with streams and channels, but the cost is that the mechanism for transmitting a variable number of arguments and co-ordinating them must be exposed.

3.1 *PMI Functions and Parallel Processing*

A PMI function can run within a separate process (though the programmer need not know that). Using this combination we can now write the samefringe program as follows:

```
(defun samefringe (t1 t2)
  (pmi-pflet t ((compare (a1 a2) (:leaf1 :leaf2)
                  (unless (eq a1 a2)
                    (return-from samefringe nil))))
    (let ((end-marker (list nil)))
      (labels ((traverse (tree arg-name)
                 (trav tree arg-name)
                 (qwait (compare arg-name end-marker)))
               (trav (tree arg-name)
                 (cond ((atom tree) (compare arg-name tree))
                       (t (trav (car tree) arg-name)
                          (trav (cdr tree) arg-name)))))
        (qprogn t
          (traverse t1 :leaf1)
          (traverse t2 :leaf2))
        t))))
```

We have used Qlisp primitives to fill in the glue portions of the code so that a complete program can be presented. We will try to eliminate those primitives with additional programming model material.

There are some strong points and some weak points of this code. The form PMI-PFLET defines a process that contains a PMI function. The arguments are named :LEAF1 and :LEAF2. The answer NIL is immediately returned if the leaves in any pair sent to COMPARE are not the same. The value of END-MARKER is used as a signal that the traversal of a tree is complete. It is used to stop the traversal of differently fringed trees when the smaller one has been traversed.

The code for TRAV is very similar to the basic traversal function presented earlier, but includes an invocation of COMPARE in which the passed-in name for the argument is used for the leaf. The code for TRAVERSE here includes the call to COMPARE with the end marker.

Finally, QPROGN creates two processes, one to traverse each tree. When both processes have finished the following is true: Each process has sent all of the atoms it found to compare. Furthermore, each has waited (using QWAIT) for the response of the COMPARE process to the end marker signal. This means that all possible pairs that COMPARE could compare have been compared, including the end markers. Therefore, if both processes have completed, the answer must be T.

There are other, interesting aspects of PMI functions that we have not discussed. A similar set of constructs can be found in (Lamping, L&FP 1988).

4. Process Environments

The above treatment of the samefringe problem is not satisfying, however. For example, naming the arguments appears to be adding too much detail to the solution—the reason for naming arguments is to distinguish arguments from different processes. In fact, passing argument names around seems like a hack rather than a clean approach. The function TRAVERSE has been distorted in order to provide a signal that a process has completed its traversal. The QPROGN guarantees that both processes have terminated before T is returned.

What we see sprinkled through this program are ad hoc mechanisms that either deal with controlling the algorithm in the presence of processes whose time-domain behavior is unpredictable or serve to direct information from one process to another using markers (here, the means of identifying processes is analogous to identifying rivers by dropping colored dyes into them).

We have searched for some solutions to this problem using an object-oriented approach. One solution is called ***process environments*** and is similar to Qlisp's heavyweight futures. A process environment is a set of processes, each of which is an instance of the class named PROCESS. The set of these processes is treated as an object that can be manipulated by prototype code. Subsets of the processes can be distinguished by their classes, and control code can be written whose behavior depends on the state of the set of processes or the states of any of the processes in a process environment. When certain important states are achieved or events occur within the environment, specific generic functions are invoked, and these generic functions may be specialized. For example, when a process terminates normally, the generic function TERMINATE is invoked by the system; its default method returns t. A process environment might correspond to an execution of existing code or of prototyping code.

Let's review the relevant parts of the Common Lisp Object System (CLOS). A generic function is a function whose implementation is as a set of methods some of which are invoked when arguments satisfy class constraints. For example, suppose there is a generic function named DISPLAY which takes two arguments, a thing to display and the display itself. Suppose further that there are graphical objects, textual objects, graphical displays, and textual displays. Then one might define display with the following four methods:

```
(defmethod display ((obj graphical-object) (disp graphical-display)) ...)
(defmethod display ((obj graphical-object) (disp textual-display)) ...)
(defmethod display ((obj textual-object) (disp graphical-display)) ...)
(defmethod display ((obj textual-object) (disp textual-display)) ...)
```

Calls to display are all syntactically similar, but which of the above methods is invoked depends on the classes of the two arguments. If OBJ is textual and DISP is graphical, the call

```
(display obj disp)
```

will invoke the third method, which is presumably customized code for that situation.

The first problem with the prototype code we can solve with a CLOS-like idea is the problem of argument naming in PMI functions. The first argument to COMPARE is just the one that came from one of the two processes and the second argument is that one that came from the other process. If values were tagged with the process from which they came, we could use that information to distinguish the arguments to COMPARE. (Note that the ability to "tag" a value with a process implies a powerful underlying mechanism. That mechanism, called ***hybrid inheritance***, will be discussed later.)

So, we define processes to be instances of classes, and we allow values to take on as an additional characteristic the class of the process that produced it. This might look like this:

```
  ...
(class-let ((p1 (process) ...)
            (p2 (process) ...))
  (pmi-flet ((compare (x:p1 y:p2) (unless (eq x y) (return-from ...))))
    ...
    (labels ((traverse (tree)
               (cond ((atom tree)
                      (compare tree))
                     (t
                      (traverse (car tree))
                      (traverse (cdr tree))))))
      (spawn p1 (traverse tree1))
      (spawn p2 (traverse tree2)))))
  ...
```

We use the notation x:c to indicate that the argument to be bound to x must be an instance of the class c. Note that this conflicts with Common Lisp package notation, which we have chosen to abandon.

At this point TRAVERSE looks about the way we want it to look. SPAWN creates an instance of a process and starts it running:

```
(spawn <class> <expression>)
```

This seems to adequately solve the problem of arguments from two different sources being delivered to a single function that requires them. The remaining problem is termination. There are two basic approaches, one using a strategy of class-based descriptive techniques and the other using monotonic variables. The monotonic variables approach will be presented in the section "Monotonic Variables."

A common technique in CLOS is to customize the behavior of a program by allowing the programmer to define methods on generic functions that are called whenever certain conditions hold or events occur. For example, in CLOS itself, if a generic function is

invoked and no methods are applicable, the generic function NO-APPLICABLE-METHOD is invoked. There is a default method for this function that signals an error, but if there is a more specific method, it will be invoked, providing customized behavior in that case.

Some of the problems with the previous samefringe solution can be addressed by providing a vocabulary for talking about processes and their activities. In some ways, this approach is akin to adding a reflective layer in which the system is able to refer to its own activities and to alter or customize them.

A natural way to think about a process is in terms of the essential nature of the state that it is in—for example, it is running, it is terminating, it has been killed, or it has terminated. CLOS provides a vocabulary for discussing an object in terms of its class. That is, an object is distinguished from others primarily by its class-oriented characteristics—the level of abstraction is the class.

As we've already seen, the first step is to define processes to be instances of classes according to the means of their creation. Some classes will be behaviorally indistinguishable from others at various levels because they are instances of the same classes.

The second step is to define processes to be instances of classes according to their state of execution. When a process is running, it is an instance of the class of running processes; when it has terminated, it is an instance of the class of dead processes.

In CLOS, every object is an instance of some class, but the class of an object changes infrequently, usually as a result of programming environment activity. Our application of these ideas is for objects to not only change (possibly) frequently, but for classes to accrete to an object. That is, some particular object may dynamically become an instance of some class for some period and then stop being an instance of that class. The object does not necessarily shed any of its previous classes during this process, but simply adds the class to the set of which it is an instance.

One ingredient of our scheme that is not yet relevant to the samefringe problem is that a process is also an instance of the class named by the function it is currently executing. This provides a mechanism for talking about complex events such as when a process is executing one function inside of another. We will see an example of the use of this later.

4.1 *Hybrid Inheritance*

CLOS multiple inheritance probably does not support this behavior very well, so we use a different sort of inheritance structure, called ***hybrid inheritance***. In hybrid inheritance, a class hierarchy is defined within a ***domain***. Within each domain, inheritance is determined by the mechanism of that domain. A domain that supports instances frequently changing class or temporarily become an instance different classes is called a ***dynamic domain***.

There are two distinct aspects of hybrid inheritance. The first aspect is that if two classes are in two different domains, an object can be a direct instance of both of them. In multiple inheritance a similar effect can be achieved by defining a class that represents

objects that are instances of those two different classes. This might be a satisfactory approach, but it is not a necessary one.

An object can also be a direct instance of two classes within a domain that supports it. The characteristics of inheritance in different domain need not be the same or similar. For example, one domain might be a CLOS-like multiple inheritance system while the other is a Smalltalk-like single inheritance system.

The second aspect is that if an object is a direct instance of two different classes, those parts of the object associated from one class are completely separate from those parts associated with the other, so that such an object is really a simple composite of two parts, each of which is entirely retained. In CLOS multiple inheritance, for example, if a class inherits from two different ones, a process of rationalization takes place that handles any name conflicts that arise. Hybrid inheritance enables a programmer to define a domain within which such rationalization takes place, but also provides a mechanism to avoid such rationalization by enabling separate parts to be pieced together. Names in such a system are meaningful only with respect to a domain.

Using hybrid inheritance it is possible for objects to become instances of a class and then to cease to be an instance of a class. This is the mechanism we use for tagging a value with the process that produced it.

Hybrid inheritance implies that objects can be simultaneously instances of two incommensurable classes. Does that make sense? Take an example from a language with abstract data types. In such a language it is possible to define a queue. The representation of a queue might be a list. Let Q be a queue. To a program that is operating on queues, Q is an instance of the type **queue**. To a debugger being used on this program, Q might be instance of the type `list`.

In this example, it might be said that the *concrete* type of Q is `list` while the *abstract* type is **queue**. We see that different programs can see the same object as different types or classes.

Another category of examples of the need to inherit from more than one object arises when orthogonal properties are being combined. For example, the property of being ***persistent*** is orthogonal to almost all other properties a data structure might have. Persistence is the property of a data structure existing beyond the lifetime of the process or program that created it. A text file in an operating system is an example of a persistent data structure. One can imagine creating a class that not only inherits from a particular family chain, but also has the property of persistence mixed in.

Hybrid inheritance is also useful for other reasons. One is that a prototype might be made up of programs written in several different programming languages. If some prototype code needs to talk about the type or class of an object within its programming language or system, there needs to be a mechanism to distinguish types in different languages. Each language is a separate domain. And an object that is of one type in one language might be of another type when passed into a program in another language.

Moreover, the behavioral or structural characteristics of a type or class system will be different from language to language, and a piece of prototyping code must be able to operate within those domains of characteristics.

When defining a method, each specializer must be a combination of domain and class. A generic function has a signature that includes the set of domains on which it is specialized. If all methods are of this same signature, the generic function behaves exactly as a CLOS generic function. If they aren't, then each signature is equivalent to a method qualifier, and method combination is used to determine how to implement an effective method composed of methods with different signatures.

When the domain is obvious, it will be elided.

4.2 *Method Applicability and Hybrid Inheritance*

In CLOS, the method that is invoked depends on the classes of the arguments to it. We extend this so that the method that is invoked also depends on the class of the process that invokes it. For example, when a process terminates, the generic function TERMINATE is invoked on no arguments. We can specialize TERMINATE by the class of the invoking process to customize the actions of the function according to which sort of process terminated.

In CLOS, a method is applicable if the arguments to the generic function are of the right classes. In the domain of processes—in which the class of a process changes dynamically—we extend the concept of applicability as follows. Suppose a generic function is invoked which is made up of methods, some of which are specialized on classes within dynamic domains. If no method is applicable at invocation time, the generic function will wait until some method is applicable. When a method becomes applicable, the processes that changed the classes of the arguments so as to cause the method to be applicable are paused until the generic function terminates. Method combination provides a mechanism to define complex behavior based on the changing states of processes. For example, one type of method combination would wait for the all arguments to be of the right class at the same time, while a second type would freeze each as it becomes the right class.

4.3 *Samefringe*

Let's look at SAMEFRINGE now:

```
 1 (defun samefringe (t1 t2)
 2   (let ((unique (list nil)))
 3     (class-let ((p1 (process) ...)
 4                 (p2 (process) ...))
 5       (pmi-pflet ((compare (x:p1 y:p2)
 6                     (unless (eq x y) (return-from samefringe nil))))
 7         (gf-flet ((terminate:process () (compare unique))
 8                   (complete (p:dead q:dead) t))
 9           (labels ((traverse (tree)
10                      (cond ((atom tree) (compare tree))
11                            (t (traverse (car tree))
12                               (traverse (cdr tree))))))
13             (complete
14               (spawn p1 (traverse t1))
15               (spawn p2 (traverse t2)))))))))
```

The special form CLASS-LET defines a set of classes with lexical scope and indefinite extent. The special form GF-FLET is similar to Common Lisp's FLET but defines a local generic function by defining its methods.

There is a class named PROCESS of which every process is an instance. Because there are two trees being traversed, there will be two subclasses of PROCESS, one for each tree, called P1 and P2. The function COMPARE will be a PMI function that takes one argument from each of these two processes. When each process terminates, TERMINATE will provide the unique end marker to COMPARE.

The main program will spawn two processes, one an instance of P1 and the other an instance of P2. That main program will then apply COMPLETE to those two processes, where the only method for COMPLETE is applicable when both its arguments are dead processes. Here DEAD is the subclass of PROCESS consisting of processes that have terminated. Therefore, COMPLETE and hence SAMEFRINGE will wait until both processes have terminated.

Line 2 defines the unique object. Lines 3–4 define two distinct subclasses of PROCESS. These subclasses only exist during the dynamic extent of the CLASS-LET. The names P1 and P2 are lexically apparent only within the lexical scope of the CLASS-LET.

Lines 5–6 define the comparison function, which compares an object (X) produced by P1 to an object (Y) produced by P2. Line 7 defines the termination procedure for the processes. When a process of class P1 terminates, the system invokes TERMINATE as if by P1, and then P1 becomes an instance of DEAD. The characteristic of being of class P1 is inherited by this invocation of TERMINATE which will pass on that tag to UNIQUE when it invokes COMPARE.

Line 8 defines a generic function that returns T when invoked on two terminated processes. Furthermore, if this function is applied to two processes, either one of which is not terminated, invocation will be blocked until they both are terminated.

Lines 9–12 are the simple traversal routine. Notice that the call to COMPARE does not manipulate argument names because the arguments are tagged by the classes (processes) that produced them. Also notice that this traversal routine is exactly the one we started with.

Lines 13–15 are the main body of the function. Lines 14–15 spawn two processes, the first an instance of P1 and the second an instance of P2. The function COMPLETE will be invoked when the two processes terminate, and this function will return T in that event. However, recall that COMPARE might terminate the processes early. One fine point is that whenever a CLASS-LET is exited, all instances are killed without invoking TERMINATE on them. (In this case a generic function named KILLED will be invoked, but the default method, which is not shown, simply returns T.)

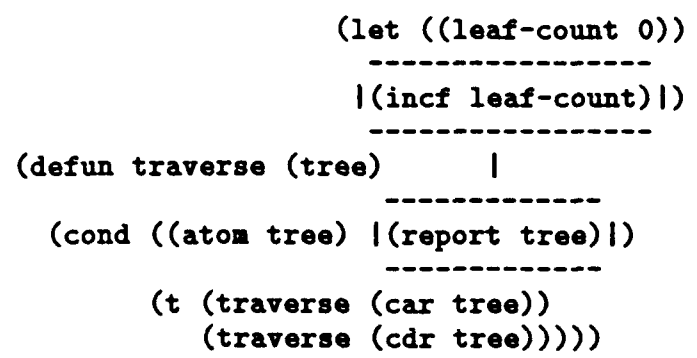
To be truthful, we still do not like this program. The use of the unique end marker still seems to be a hack, even though one could argue that it is nothing more that an end-of-stream marker. Later we will present another way of programming this function using monotonic variables, which might be a little better. Note also that we could have used some sort of stream abstraction to do the same thing. The prototyping language will probably also have such abstractions, and the above program is not intended to be the only or even best way to program the solution to this problem.

5. Illuminated Code

What we've seen is the textual endpoint of a prototyping activity. This endpoint is necessary to enable tools that operate on source code to have access to prototype code. The environment portion of the prototyping system is largely language-independent and tool-independent, as long as those languages and tools obey a particular set of protocols. We want source-level tools to operate within this framework.

We are working on a partially visual, mostly textual means of creating this prototype from existing code.

Illuminated code is code that is annotated with various environmental and linguistic constructs or notes that cause information to be gathered, moved about, or program structures to be created. For example, the following code is illuminated to count the number of leaves found during traversal:



One interesting side point on this example is that the placement of the initialization of LEAF-COUNT to 0 is not important. It is important that it have been initialized outside the scope of TRAVERSE and the fragment that increases LEAF-COUNT. Note that the structure of the annotation implies these constraints. Let's quickly look at the full code to do something like this in Lisp:

```
(let ((leaf-count 0))
  (let ((increase-leaf-count #'(lambda () (incf leaf-count))))
    (defun traverse (tree)
      (cond ((atom tree)
             (progn (funcall increase-leaf-count)
                    (report tree)))
            (t (traverse (car tree))
               (traverse (cdr tree)))))))
```

We needed to specify several irrelevant things here. First is the name INCREASE-LEAF-COUNT. Second is the placement of the initialization of LEAF-COUNT. Third is the insertion of the call to INCREASE-LEAF-COUNT in TRAVERSE. The fourth is the inclusion of the definition in the nested LET expressions so that the visibility of the irrelevant name INCREASE-LEAF-COUNT would be correct. We could have avoided some of this by using an advise tool to modify the definition of REPORT to call INCREASE-LEAF-COUNT. Of course, we would need to make sure such advice did not conflict with other advice.

Annotations make the description of the essential parts of such additions easier, and a simple transformation system can chose a textual implementation and specify the irrelevant.

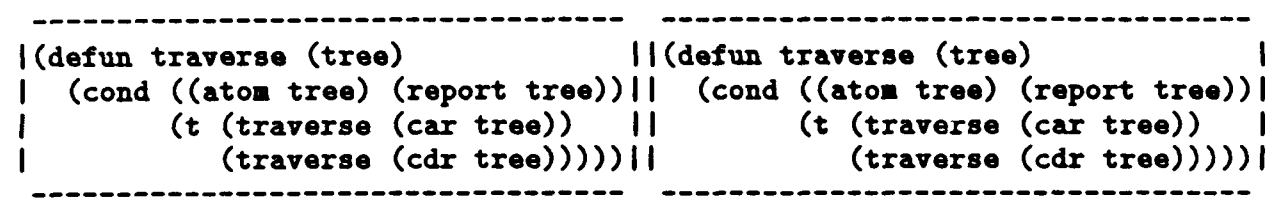
In general, the annotation/abbreviation model enables programs to be modified and viewed in different ways without destroying the original programs and without resorting to bulky versioning systems.

Here is the code we assume we're starting with to work on a prototype of SAME-FRINGE.

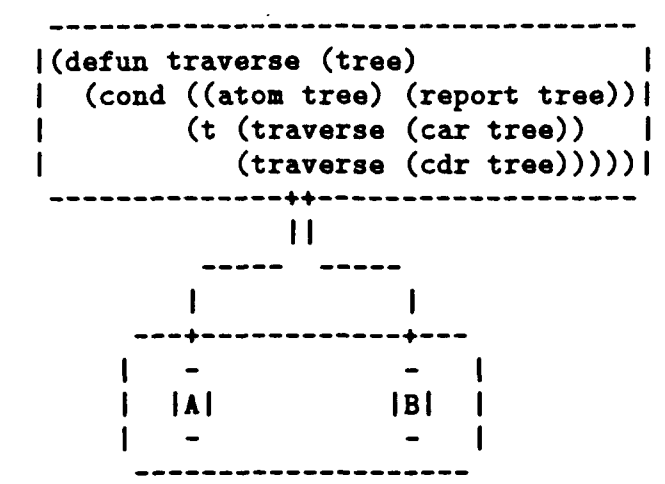
```
(defun traverse (tree)
  (cond ((atom tree) (report tree))
        (t (traverse (car tree))
           (traverse (cdr tree)))))
```

Using the annotation/abbreviation model environment, we would create a ***palette*** with two objects on it: Each is an annotation whose abbreviation is just the above code, and the annotation simply points to the configuration that holds that original code. The class of the annotation indicates that each of the annotations on the palette is to be regarded as a copy if it is modified. A palette is nothing more than a very large window that contains code for a particular prototype written in the prototyping language. As such it must be able to manipulate source and graphics, and its contents are to rendered into textual form. Every palette implicitly represents a process environment.

Here is what the palette looks like:



Below is a schematic form of the above situation. The definition of TRAVERSE is someplace in a configuration of other source code. There are two annotations on it, one linking to the object labeled A and the other linking to the object labeled B. A and B are on the palette, which is indicated by the box surrounding the boxes labeled A and B. A and B are displayed using the abbreviation that is simply the original source code. That is, the two boxes of code just above are the abbreviated display of the boxes labeled A and B below:



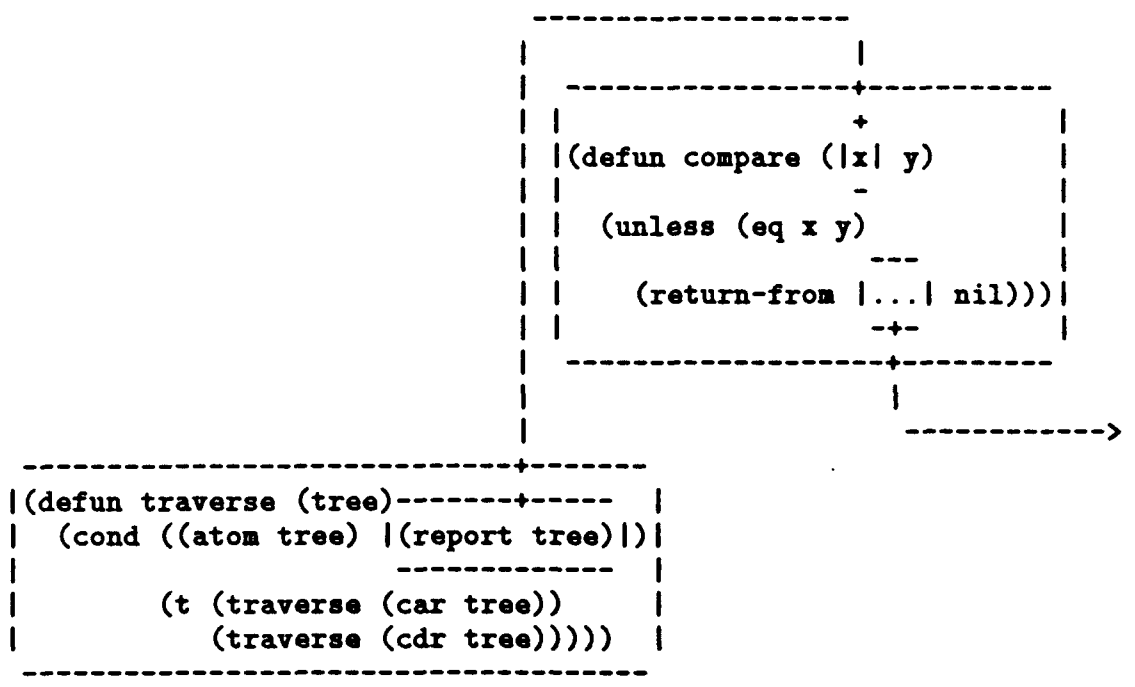
Looking back at the palette, we annotate each box with a note stating it is an instance of a class. This accomplishes stating that there are two subclasses of PROCESS, P1 and P2. We need not name them.

The palette represents the SAMEFRINGE prototype, and we now define COMPARE on it:

```
(defun compare (x y)
  (unless (eq x y)
    (return-from ... nil)))
```

We annotate the argument X in the argument list in COMPARE with the form (REPORT TREE) in the left-hand instance of TRAVERSE, and the argument Y with the form (REPORT TREE) in the right-hand instance of TRAVERSE. Because the two instances are distinguished by the classes of the processes they are instances of, this is enough to generate the PMI definition of COMPARE. We annotate the ellipsis with the palette, which implies that if X and Y are not EQ, NIL is returned from the computation defined by the code on the palette.

It's hard to show this without fancy graphics, but it would be something like this, using only the annotation on X as an example:



Now we write down the definition of COMPLETE:

```
|(defun complete (p1 p2) t)|
```

The arguments are similarly annotated with the processes. Finally, TERMINATE:

```
|(defun terminate () (compare unique))|
```

This code is annotated with the two instances of TRAVERSE. Because TERMINATE is invoked by the system when the processes on the palette terminate, it is necessary only to place it on the palette with the annotation. Probably the remainder of the program can be completed automatically. The annotation is an instance of the class DEAD, which indicates that TERMINATE is invoked when the corresponding process dies.

The finished code seen in the last section could be generated from the palette and annotations. The palette is annotated with that finished code. The various parts of the code and palette are cross-annotated.

The traversal code we started with is exactly what we needed to incorporate into the prototype with minimal changes. Suppose we were not so well off and that the code we wanted to start with looked like this:

```
(defun traverse (tree)
  (operate tree)
  (unless (atom tree) (traverse (car tree)) (traverse (cdr tree))))
```

This code traverses trees exactly the right way, but it invokes an inappropriate operation at every node whether internal or leaf. To use this code we have several options:

1. Annotate the expression (OPERATE TREE) with this code:

```
(when (atom tree) (compare tree))
```

   The effect of this is to alter the original code to be as follows:

```
(defun traverse (tree)
  (when (atom tree) (compare tree))
  (unless (atom tree) (traverse (car tree)) (traverse (cdr tree))))
```

   The use of an annotation has the following effect. The actual code that is executed is as it appears immediately above. That code is derived from the original source code by substituting the destination of the annotation (the WHEN expression) for the source of the annotation (the OPERATE expression). However, the original source for TRAVERSE remains in the configuration as it always has, and only in the "mind of an execution engine" does the modified code exist. When the prototype is complete, the final code can be automatically derived by looking through the layers of annotations. In this case, we could further annotate this line and the following with a transformation that would collapse the WHEN followed by the UNLESS into a more compact conditional expression.
2. Annotate TRAVERSE with this;

```
(class-let ((atomic-tree (predicated-class) #'atom))
  (gf-labels ((operate (tree:atomic-tree) (compare tree))
              (operate (tree:t) nil))
    (defun traverse (tree)
      (operate tree)
      (unless (atom tree) (traverse (car tree)) (traverse (cdr tree))))))
```

   Here the idea is to slightly alter the context of execution (or compilation) so that when the original program called OPERATE, the prototype conditionally invokes COMPARE. A predicated class is one whose instances satisfy some predicate. If the behavior of OPERATE must be preserved, there might be more we need to do with this approach.
3. Introduce and invoke a generic function that continually waits for the original program to achieve a complex state:

```
(defgeneric watch-visit :method-combination :continuous)
(defclass atomic-tree (predicated-class) () :predicate #'atom)
(defmethod watch-visit (p:operate<tree:atomic-tree>) (compare tree))
```

Here the method combination type states that the generic function is invoked continually. The function WATCH-VISIT must be invoked on the process that traverses the trees:

```
(defun samefringe (t1 t2)
  ...
        (labels ((traverse (tree)
                   (operate tree)
                   (unless (atom tree)
                     (traverse (car tree))
                     (traverse (cdr tree)))))
          (complete
            (watch-visit (spawn p1 (traverse t1)))
            (watch-visit (spawn p2 (traverse t2))))))
```

By "continually" we mean that when WATCH-VISIT is applicable, the process that caused it to be applicable is paused and WATCH-VISIT is completed. Once WATCH-VISIT is completed, the previously paused process is continued until it would no longer be applicable. At that point WATCH-VISIT is re-invoked. That is, the order of events are as if a process, say P1, that is traversing a tree reaches OPERATE. The process P1 is paused and WATCH-VISIT runs. When WATCH-VISIT ends, P1 is continued until it exits OPERATE. At that point WATCH-VISIT is re-invoked (and P1 continues as well).
The odd syntax means this: WATCH-VISIT will be called when the process P is within OPERATE with a first argument of class ATOMIC-TREE. Note that the argument to OPERATE is available in the WATCH-VISIT method.
This could also have been written as follows, where applicability and pausing are as described above.

```
(defclass atomic-tree (predicated-class) () :predicate #'atom)
(defmethod watch-visit (p:operate<tree:atomic-tree>)
  (compare tree) (watch-visit p))
```

4. Introduce a generic function that will be triggered when OPERATE is invoked with a first argument of class ATOMIC-TREE:

```
(defclass atomic-tree (predicated-class) () :predicate #'atom)
(defmethod watch-visit:<tree:atomic-tree> :triggered () (compare tree))
```

where SAMEFRINGE looks like this:

```
(defun samefringe (t1 t2)
  ...
        (labels ((traverse (tree)
                   (operate tree)
                   (unless (atom tree)
                     (traverse (car tree))
                     (traverse (cdr tree)))))
          (watch-visit)
          (complete
            (spawn p1 (traverse t1))
            (spawn p2 (traverse t2)))))
```

We will discuss triggered functions more later.

6. Naming Variables

When dealing with the internal parts of one program from another, we need to be able to refer to the variables of one program from another. This requires a means to name variables to distinguish them from variables of the same name in other programs. We can implement program-specific variables by considering their names to be qualified by the process in which the program defining them is being executed. Similarly if the need arose to introduce new variables, the same mechanism could be used. Here's an example:

```
(class-let ((p1 (process)) (p2 (process)))
  (let ((n:p1 0)(n:p2 0))
    (flet ((count (l) (dolist (i l) (incf n))))
      (spawn p1 (count ...))
      (spawn p2 (count ...)) ...)))
```

7. Monotonic Variables

Another useful concept is called "monotonic variables." A monotonic variable is one whose value always increases or always decreases. Subclasses of monotonic variables are MONOTONIC:INCREASING and MONOTONIC:DECREASING.

Here is an interesting generic function:

```
(defmethod lesser (x:dead y:dead)
  (cond ((< x y) t)
        (t nil)))

(defmethod lesser (x:dead: y:monotonic:increasing)
  (cond ((< x y) t)
        (t (lesser x y))))

(defmethod lesser (x:monotonic:increasing y:dead)
  (cond ((<= y x) nil)
        (t (lesser x y))))
```

The idea is that if one of the numbers is dead and the other monotonic increasing, we can sometimes know which is smaller before both objects are dead. This can then be used to terminate processes at appropriate points. We can define similar methods for other combinations of DEAD, MONOTONIC:INCREASING, and MONOTONIC:DECREASING.

For example, here is a simple function to find the shorter of two lists:

```
(defun shorter (l1 l2)
  (class-let ((p1 (process)) (p2 (process)))
    (let ((n:p1:monotonic:increasing 0)(n:p2:monotonic:increasing 0))
      (flet ((len (l)(dolist (x l) (incf n))))
        (spawn p1 (len l1))
        (spawn p2 (len l2))
        (cond ((lesser n:p1 n:p2) l1)
              (t l2))))))
```

For example, if L1 is shorter than L2, N:P1 will possibly be computed and rendered dead before P2 completes. If this happens, LESSER will terminate when N:P2 exceeds N:P1, which will exit the CLASS-LET defining P1 and P2, which will kill P2. The use of monotonic variables prevents SHORTER from running for a long time when the answer can be more quickly determined.

We can write a different version of SAMEFRINGE using monotonic variables:

```
(defun samefringe (t1 t2)
  (class-let ((p1 (process) ...)
              (p2 (process) ...))
    (pmi-pflet ((compare (x:p1 y:p2)
                  (unless (eq x y) (return-from samefringe nil))))
      (let ((n:p1:monotonic:increasing 0)(n:p2:monotonic:increasing 0))
        (gf-flet ((count:compare:process :trigger () (incf n)))
          (labels ((traverse (tree)
                     (cond ((atom tree) (compare tree))
                           (t (traverse (car tree))
                              (traverse (cdr tree))))))
            (spawn p1 (traverse t1))
            (spawn p2 (traverse t2))
            (= n:p1 n:p2)))))))
```

Here, the function COUNT is a triggered function: When some process becomes an instance of an invocation of COMPARE, that process is paused while the body of COUNT is executed. That invocation of COUNT accretes the classes of the process that triggered it, so that INCF increases the right variable.

In the main body, SAMEFRINGE spawns the two processes and then returns the result of the equality test, which is implemented similarly to LESSER above.

8. Temporal or Historical Abstraction

Some of the problems in prototyping have to do with modifying a program to keep track of the history of a data structure or to answer questions about what things happened at what time. We believe there is a new type of abstraction, called ***temporal*** or ***historial abstraction*** that can make such changes simpler.

Consider a program that is simulating marriage and employment for the purposes of understanding how government policy decisions should be made about affirmative action. In such a simulation one could imagine a class being defined whose instances represent people. These instances might have some structure (a data abstraction) and a protocol.

One slot in the instance might store the gender of the individual and another might record whether the person is married. Part of the protocol is an initialization of an instance to store the correct gender, and another part is to assign the correct setting for the marriage slot during the abstract operations of marriage, divorce, and spousal death.

From the point of view of a reader of this simulation program, these two slots are identical except insofar as what is stored in them. Yet, from a real-world semantics point of view, they represent incommensurable things: The gender of an individual is a characteristic of the individual, and so it makes sense for the gender slot to be part of the representation of the individual. The marriage slot is a characteristic about the history of the individual, yet the marriage slot is not part of the representation of the history of the

individual. From a program semantics point of view the value of the marriage slot depends exactly on the history of the representation of the individual.

Therefore, the use of a data abstraction to represent whether an individual is married is incongruous regardless of which of the two most plausible ways you look at it.

If we later decide that the simulation needs to depend on whether a person was previously divorced, a new slot might be added to the class so that each individual records whether they have been previously married. There are many such historical questions we wish to ask of objects in the simulation: how many times was someone married, was this child born before or after the second divorce, was the salary of a person ever larger than the salary of the spouse.

The proliferation of data abstractions to implement other abstractions in this example leads one to wonder how many data abstractions are really implementations of different types of abstraction.

Even if the need to define a data abstraction to store historical information were removed, the problem of temporal abstractions would not be solved if the programmer still were required to insert protocol invocations at the points at which the history needed to be updated. This would defeat locality. Furthermore, it would reveal implementation.

Because a prototyper wishes to modify a program as little as possible, a temporal or historical abstraction mechanism would be welcome, if it were then possible to isolate them from the original code.

We believe that our class-based description mechanisms are a step in this direction. For example, a triggered function defines a temporal or historical abstraction, though in a primitive way. We wish to define a common pattern that captures the essence of counting the number of times that a process enters a particular state. Usually one would have to define some storage and then modify occurrences of the lexical manifestation of the target state to invoke code to update that storage to accomplish this, but just as one hopes to isolate the implementation of data abstractions from their use, so one hopes to isolate the implementation of the historical abstraction from its use.

In this case, its use is its definition, and its implementation is how the definition is turned into the proper bookkeeping code.

In some sense, this is just a highfalutin way of saying that with historical abstractions, we do not need to modify TRAVERSE to get SAMEFRINGE (in this case such modification is easy, but with more complex prototypes it might not be so easy):

```
(defun samefringe (t1 t2)
  (class-let ((p1 (process) ...)
              (p2 (process) ...))
    (pmi-pflet ((compare (x:p1 y:p2)
                  (unless (eq x y) (return-from samefringe nil))))
      (let ((n:p1:monotonic:increasing 0)(n:p2:monotonic:increasing 0))
        (labels ((traverse (tree)
                   (cond ((atom tree) (incf n) (compare tree))
                         (t (traverse (car tree))
                            (traverse (cdr tree))))))
          (spawn p1 (traverse t1))
          (spawn p2 (traverse t2))
          (= n:p1 n:p2))))))
```

These two versions have several nice properties. First, each is pretty concise. Second, each matches well the description of the solution to the problem, repeated here:

> First we want to set up two processes, one to traverse T1 and the other to traverse T2. Then we set up a third process to look at the atoms coming from each traversal. If any two corresponding atoms are not the same, the processes are terminated and the answer is nil. Otherwise, if both traversal processes end normally, the two trees have the same fringe if both processes terminated after reporting the same number of atoms.

Some bad properties are the ugly, visible definitional forms: CLASS-LET, PMI-PFLET, LET, and GF-FLET, which can be hidden with annotations and abbreviations. The use of a visual layer on top of simpler text would eliminate some of this—remember that this is simply the textual representation of the program.

9. Conclusions

It's probably hard to tell what is really going on from this precis, but that's how research goes.

Ten Ideas for Programming Language Design

Richard P. Gabriel
Stanford University

Over the last several years I have been working on prototyping, and during that time I worked a little on ten ideas for programming language design. Since then I have shifted my focus to environment concerns, principally on providing mechanisms for expressing program composition and modification without altering the underlying programming language—essentially a layer of linguistic support for prototyping within the layer of environment and above the layer of language.

These ten ideas may form part of a prototyping system, but they are also relevant to programming language design, I don't want these ideas to slip away, so this paper presents a short introduction to each of them along with some examples of their expressive power in the hope that someone might wish to pick one up and work with it a bit. Enjoy.

The code examples I present are in a parallel language that is a cross between an unknown language and a dialect of Lisp blending Common Lisp, Qlisp, and Scheme.

1. Idea 1: Hybrid Inheritance

The first idea is called ***hybrid inheritance***, and it is used in the rest of the paper. The key to hybrid inheritance is the ***domain***. A domain is a set of classes, and the domains form a partition of all classes. Each domain has its own inheritance, instantiation, subclassing, and method applicability or inheritance scheme. If one class is a subclass of another, both are in the same domain.

An object can be an instance of classes in different domains. Such an object is the disjoint union of subobjects, each of which is an instance of a class from one of the domains. In CLOS, CLASS-OF takes an object and returns the unique class of which it is an instance. With hybrid inheritance, the analog of CLASS-OF takes an object and a domain, and returns the class in that domain of which it is an instance. Similarly, SLOT-VALUE takes a slot name, an object, and a domain. If there is a domain named T, it might make sense for the domain to default to it.

There will be function that could be called COMPOSE-INSTANCE that takes two instances of classes in domains that can be combined, and combines them into one instance:

```
(compose-instance <instance> <instance>)
```

The work was supported by DARPA, contract number N00039-84-C-0211.

Another function would be one to take an existing instance and make it be an instance of a class in a compatible new domain:

```
(make-be-instance <instance> <class> ...)
```

This makes `<instance>` be an instance of the supplied class. The elided expressions are initialization arguments used to construct the new subcomponent.

There are several motivations for hybrid inheritance:

- In a prototyping system in which several languages are simultaneously supported, it is important to have objects that represent the same thing in different languages, each with its own type or class system. For examp' a prototyping system might wish to represent an object that is an instance of a Smalltalk class when that object is manipulated by Smalltalk code, and an instance of a CLOS class when manipulated by Common Lisp code.
- In a prototyping system, it is important to be able to distinguish representational or concrete types from abstract ones. For example, an object might be a node in a graph to the abstract code and a vector-like structure to a debugger.
- Most importantly, a much more expressive prototyping language can be designed if it is possible to create objects that can take on additional types as markers during execution. For example, it is useful to be able to tag an object with the process that produced it. In a prototyping system, processes might be instances of classes in a process domain, and an object is an instance of a class representing a process if that object was created or manipulated by that process. This ability to accrete classes will be important later.

CLOS-like generic functions will be used in the rest of the paper. There is a problem with this choice: In general methods must be ordered by specificity, and if two methods specialize on an argument using classes from different domains, how should they be ordered? The same problem is faced with methods with different method qualifiers, and the solution is the same: neither is more specific than the other. In fact, they are only partially ordered by specificity. Methods with different domain signatures are combined using method combination.

I will use the syntax `x:c` to indicate that the object named by `x` has quality `c`. In most cases this will be the same as saying that the object named by `x` will be an instance of the class named `c`. The syntax `x:c&d` indicates that the object named by `x` has the qualities `c` and `d`.

2. Idea 2: Process Environments

A ***process environment*** is a set of processes, each of which is an instance of the class named PROCESS. The set of these processes is treated as an object that can be manipulated by prototype code. Subsets of the processes can be distinguished by their classes, and control code can be written whose behavior depends on the state of the set of processes or the states of any of the processes in a process environment. When certain important states are achieved or events occur within the environment, specific generic functions are invoked, and these generic functions may be specialized. For example, when a process terminates normally, the generic function TERMINATE is invoked by the system; its default method returns T. We will see an example of this later.

A process environment might correspond to an execution of existing code or of prototyping code.

```
(class-let ((p1 (process))
            (p2 (process)))
  (spawn p1 (traverse tree1))
  (spawn p2 (traverse tree2)))
```

This code fragment creates two subclasses of PROCESS. The classes P1 and P2 have lexical scope and indefinite extent. SPAWN takes a class and an expression, and creates an instance of the class that will execute the expression.

There is an interesting wrinkle: A value returned from a process environment is made an instance of the process that originated it. I call any domain that changes, acquires, or sheds classes like this a ***dynamic domain***.

```
(class-of
  (class-let ((p (process)))
    (spawn p (traverse tree)))
  process-domain) => <class ...>
```

If an object is passed through a process—by being bound to a variable, for example—the object becomes an instance of that process. The effect is that an object retains the history of its creation and manipulation by retaining the classes of which it has been an instance. Note that this behavior is supplied by the domain of processes.

Since there is no particular magic to processes, I extend this idea to functions: All functions are viewed as classes, and a value returned by a function is an instance of that function (viewed as a class). This leads to the idea of superclasses of functions that would have some common traits. An obvious idea is to create classes of functions with scheduling constraints: For example, an instance of some class is such that only one such instance can be running at a time, or a function of a certain type can be run in parallel as long as the scheduler allows it.

The choice of making functions being classes will make more sense once we see the use of generic functions to trigger behavior when the state of a system becomes something described by a set of classes.

3. Idea 3: Improved Partially, Multiply Invoked Functions

The basic idea of partially, multiply invoked functions (PMI Functions) is to separate the process of coordinating the arrival of arguments from the process of executing the function on those arguments.

All calls to a function go through an interface to the actual implementation. The implementation of the function receives arguments by position, while the interface accepts only named arguments, provides for all defaulting, and coordinates the arrival of arguments for the function from multiple sources.

The Qlisp work produced the basic idea of PMI functions in late 1987 [ref: Goldman] but the Qlisp formulation has a serious drawback. Namely, arguments need to have names, which often requires those names to be passed about as additional arguments so that the proper values were assigned to the proper parameter names.

This technique is similar to currying functions, but because all arguments to the interface are named, one does not need to curry in any particular order. One could term it ***dynamic currying***. All calls to a PMI function that supply arguments to the same invocation receive the same future as their value. A future is returned whenever some required arguments to the function have not been supplied to the interface by a function call. If a particular invocation of a function has returned a future, the value returned when all required arguments have been supplied is a realized future. This is to preserve EQ-ness of all values returned for a particular invocation.

The improvement is to use hybrid inheritance. Here is an example:

```
(defclass p1 (process) ...)
(defclass p2 (process) ...)

(pmi-qdefun add-up (x:p1 y:p2) (+ x y))

(add-up (spawn p1 1)) => <future>
(add-up (spawn p2 2)) => <the same future> = 3
```

The first call to ADD-UP supplies the argument of type P1, and the second the argument of type P2. These two are added to produce the result. Another example is this:

```
(defun f1 () 1)
(defun f2 () 2)

(pmi-qdefun add-up (x:f1 y:f2) (+ x y))
```

```
(add-up (f1)) => <future>
(add-up (f2)) => <the same future> = 3
```

4. Idea 4: Discriminating on the Calling Process or Function

In CLOS we can discriminate on any of the (required) arguments. When processes are classes it makes sense to discriminate on the class of the caller. The following program prints P1 when the traversal of TREE1 terminates and P2 when the traversal of TREE2 terminates. Note that this uses the automatic invocation of TERMINATE in process environments.

```
(class-let ((p1 (process))
            (p2 (process)))
  (gf-labels ((terminate:p1 () (print 'p1))
              (terminate:p2 () (print 'p2)))
    (spawn p1 (traverse tree1))
    (spawn p2 (traverse tree2))))
```

We defined two methods for TERMINATE, one invoked when called by P1 and one when called by P2. GF-LABELS defines methods and is similar in spirit to GENERIC-LABELS in CLOS. CLASS-LET creates a process environment.

5. Idea 5: Waiting for Methods to be Applicable

We've seen it's possible to write methods that are invoked if arguments are instance of process classes or function classes. In a multiprocessing environment, it can happen that a generic function might be invoked that will have an applicable method at some point, but not at the time of invocation.

In CLOS, a method is applicable if the arguments to the generic function are of the right classes. In the domain of processes—in which the class of a process changes dynamically—we extend the concept of applicability as follows. Suppose a generic function is invoked which is made up of methods, some of which are specialized on classes within dynamic domains. If no method is applicable at invocation time, the generic function will wait until some method is applicable. When a method becomes applicable, the processes that changed the classes of the arguments making the method applicable are paused until the generic function terminates. Method combination provides a mechanism to define complex behavior based on the changing states of processes. For example, one type of method combination would wait for the all arguments to be of the right class at the same time, while a second type would freeze each as it becomes the right class.

Here is an example:

```
(class-let ((p1 (process))
            (p2 (process)))
  (gf-labels ((complete (p:dead q:dead) (print 'both-dead)))
    (complete
      (spawn p1 (traverse tree1))
      (spawn p2 (traverse tree2)))))
```

When a process terminates, it becomes an instance of the class DEAD. Here the method for COMPLETE will eventually be applicable, but possibly not at the moment it is invoked.

Not all generic functions have this behavior, only a subclass of them. For example, we might want to define a generic function and a set of mutually exclusive methods so that exactly one is applicable at a time, but we want that one to be invoked right at the instant of generic function invocation.

Now we are at the point where we can write a complete program that shows off all the ideas so far. This program is samefringe:

```
 1 (defun samefringe (t1 t2)
 2   (let ((unique (list nil)))
 3     (class-let ((p1 (process))
 4                 (p2 (process)))
 5       (pmi-qflet ((compare (x:p1 y:p2)
 6                     (unless (eq x y) (return-from samefringe nil))))
 7         (gf-flet ((terminate:process () (compare unique))
 8                   (complete (p:dead q:dead) t))
 9           (labels ((traverse (tree)
10                      (cond ((atom tree) (compare tree))
11                            (t (traverse (car tree))
12                               (traverse (cdr tree))))))
13             (complete
14               (spawn p1 (traverse t1))
15               (spawn p2 (traverse t2)))))))))
```

The special form CLASS-LET defines a set of classes with lexical scope and indefinite extent. The special form GF-FLET is similar to GF-LABELS

Because there are two trees being traversed, there will be two subclasses of PROCESS, one for each tree, called P1 and P2. The function COMPARE will be a PMI function that takes one argument from each. When each process terminates, TERMINATE will provide the unique end marker to COMPARE.

The main program will spawn two processes, one an instance of P1 and the other an instance of P2. That main program will then apply COMPLETE to those two processes, where the only method for COMPLETE is applicable when both its arguments are dead processes. Therefore, COMPLETE and hence SAMEFRINGE will wait until both processes have terminated.

Line 2 defines the unique object. Lines 3–4 define two distinct subclasses of PROCESS. These subclasses only exist during the dynamic extent of the CLASS-LET.

Lines 5–6 define the comparison function, which compares an object produced by P1 to an object produced by P2. Line 7 defines the termination procedure for the processes. When a process of class P1 terminates, the system invokes TERMINATE as if by P1 and then P1 becomes an instance of DEAD. The characteristic of being of class P1 is inherited by this invocation of TERMINATE which will pass on that tag to UNIQUE when it invokes COMPARE.

Line 8 defines a generic function that returns T when invoked on two terminated processes. Furthermore, if this function is applied to two processes, either one of which is not terminated, invocation will be blocked until they both are terminated.

Lines 9–12 are the simple traversal routine.

Lines 13–15 are the main body of the function. Lines 14–15 spawn two processes, the first an instance of P1 and the second an instance of P2. The function COMPLETE will be invoked when the two processes terminate, and this function will return T in that event. However, recall that COMPARE might terminate the processes early. One fine point is that whenever a CLASS-LET is exited, all instances are killed without invoking TERMINATE on them. (In this case a generic function named KILLED will be invoked, but the default method, which is not shown, simply returns T.)

6. Idea 6: Triggered Functions

Sometimes we wish to write some code that will run when certain events take place, but we wish the description of these events to be more than a simple interrupt or trap.

Suppose we are working on a prototype of the samefringe problem just described, given the following definition of TRAVERSE, which we wish to leave as is:

```
(defun traverse (tree)
  (operate tree)
  (unless (atom tree) (traverse (car tree)) (traverse (cdr tree))))
```

This code traverses trees exactly the right way for samefringe, but it invokes OPERATE at every node whether internal or leaf, so we cannot simply redefine it to do what we want. We can transform this into the right program several ways, but one of the more interesting ways is with triggered functions.

A triggered function continually waits for the original program to achieve a complex state:

```
(defclass atomic-tree (predicated-class) () :predicate #'atom)
(defgeneric watch-visit :method-combination :continuous)
(defmethod watch-visit (p:(operate tree:atomic-tree)) (compare tree))
```

Here the method combination type indicates continuous invocation. The function WATCH-VISIT is invoked on the tree traversal process, and the method shown is applicable when the argument to OPERATE is an atomic tree:

```
(defun samefringe (traverse t1 t2)
  (let ((unique (list nil)))
    (class-let ((p1 (process))
                (p2 (process)))
     (pmi-qflet ((compare (x:p1 y:p2)
                    (unless (eq x y) (return-from samefringe nil))))
       (gf-flet ((terminate:process () (compare unique))
                 (complete (p:dead q:dead) t))
           (complete
             (watch-visit (spawn p1 (traverse t1)))
             (watch-visit (spawn p2 (traverse t2)))))))))
```

By "continually" I mean that when WATCH-VISIT is applicable, WATCH-VISIT is invoked and then immediately reinvoked; however, there is only one invocation of WATCH-VISIT per triggering event. That is, if an event invokes WATCH-VISIT, WATCH-VISIT cannot be reinvoked by that same event—the event must terminate first, at which point an event exactly like the first one can invoke WATCH-VISIT.

This behavior can be implemented as follows. The process that caused it to be applicable is paused and WATCH-VISIT is completed. Once WATCH-VISIT is completed, the previously paused process is continued until the method would no longer be applicable. At that point WATCH-VISIT is re-invoked. For example, the order of events is this: process P1 begins to invoke OPERATE on an atomic tree; P1 is paused and WATCH-VISIT runs. When WATCH-VISIT ends, P1 is continued until it exits OPERATE. At that point WATCH-VISIT is re-invoked (and P1 continues as well).

The odd syntax means this: WATCH-VISIT will be called when the process P is within OPERATE with a first argument of class ATOMIC-TREE. The argument to OPERATE is available in the WATCH-VISIT method, but is not used in this example.

This could also have been written as follows, where applicability and pausing are as above.

```
(defmethod watch-visit (p:(operate tree:atomic-tree))
  (compare tree) (watch-visit p))
```

Another variation is to introduce a generic function that will be triggered when OPERATE is invoked with a first argument of class ATOMIC-TREE:

```
(defmethod watch-visit:(operate tree:atomic-tree)
  :triggered () (compare tree))
```

SAMEFRINGE would be exactly as above except there is no explicit invocation of WATCH-VISIT.

Triggered functions are but abstraction: We identified a common pattern of behavior and defined a stylized means to express that pattern.

A simple version of this behavior can be implemented by using :after methods in CLOS. For example:

```
(defmethod operate :after ((tree atomic-tree))
  (compare tree))

(defun samefringe (traverse t1 t2)
  (let ((unique (list nil)))
    (class-let ((p1 (process))
                (p2 (process)))
     (pmi-qflet ((compare (x:p1 y:p2)
                   (unless (eq x y) (return-from samefringe nil))))
       (gf-flet ((terminate:process () (compare unique))
                 (complete (p:dead q:dead) t))
           (complete
             (spawn p1 (traverse t1))
             (spawn p2 (traverse t2)))))))))
```

There are two differences. First, using ordinary method combination requires knowing a *generic* function to combine with. The sort of triggering we want is to trigger on a class of generic functions or events. We should be able define a class of generic functions—possibly an ad hoc collection—and trigger when any of them are invoked.

Second, using standard method combination requires both the generic function in question to use standard method combination and knowing that it does. User defined method combination might be incompatible with attaching methods without detailed knowledge of that method combination type.

7. Idea 7: Naming Variables Names

When piecing together programs from existing parts, we need to be able to refer to the variables of one program from another. This requires a means to additionally name variables to distinguish them from variables of the same name in other programs. We can implement program-specific variables by qualifying their names with the process in which the program defining them is being executed. Similarly if the need arose to introduce new variables, the same mechanism could be used. Here's an example:

```
(class-let ((p1 (process)) (p2 (process)))
  (let ((n:p1 0)(n:p2 0))
    (flet ((count (l) (dolist (i l) (incf n))))
      (spawn p1 (count ...))
      (spawn p2 (count ...)) ...)))
```

This program computes the length of two lists into two copies of the variable n.

8. Idea 8: Monotonic Variables

Another useful concept is called "monotonic variables." A monotonic variable is one whose value always increases or always decreases. This basic idea has been introduced elsewhere, but my use of it with classes is new. There is a class MONOTONIC and two subclasses, MONOTONIC-INCREASING and MONOTONIC-DECREASING.

Here are a few methods for an interesting generic function:

```
(defmethod lesser (x:dead y:dead)
  (cond ((< x y) t)
        (t nil)))

(defmethod lesser (x:dead: y:monotonic-increasing)
  (cond ((< x y) t)
        (t (lesser x y))))

(defmethod lesser (x:monotonic-increasing y:dead)
  (cond ((<= y x) nil)
        (t (lesser x y))))
```

The predicate LESSER can be decided before both values are computed if the process computing one is dead and the other is monotonic increasing. This can then be used to terminate processes early. We can complete LESSER by defining similar methods for other combinations of DEAD, MONOTONIC-INCREASING, and MONOTONIC-DECREASING.

For example, here is a simple function to determine whether one list is shorter than another:

```
(defun shorter (l1 l2)
  (class-let ((p1 (process)) (p2 (process)))
    (let ((n:p1&monotonic-increasing 0)
          (n:p2&monotonic-increasing 0))
      (flet ((len (l)(dolist (x l) (incf n))))
        (spawn p1 (len l1))
        (spawn p2 (len l2))
        (lesser n:p1 n:p2)))))
```

For example, if L1 is shorter than L2, N:P1 will possibly be computed and rendered dead before P2 completes. If this happens, LESSER will terminate when N:P2 exceeds N:P1, which will exit the CLASS-LET defining P1 and P2, which will kill P2.

We can write a different version of SAMEFRINGE using monotonic variables:

```
(defun samefringe (t1 t2)
  (class-let ((p1 (process))
              (p2 (process)))
    (pmi-qflet ((compare (x:p1 y:p2)
                  (unless (eq x y) (return-from samefringe nil))))
      (let ((n:p1&monotonic-increasing 0)
            (n:p2&monotonic-increasing 0))
        (gf-flet ((count:compare :trigger () (incf n)))
          (labels ((traverse (tree)
                     (cond ((atom tree) (compare tree))
                           (t (traverse (car tree))
                              (traverse (cdr tree))))))
            (spawn p1 (traverse t1))
            (spawn p2 (traverse t2))
            (= n:p1 n:p2)))))))
```

Here, the function COUNT is a triggered function: When some process becomes an instance of an invocation of COMPARE, that process is paused while the body of COUNT is executed. That invocation of COUNT retains the classes of the process that triggered it, so that INCF increases the right variable. The function = is written as LESSER was.

9. Idea 9: Temporal or Historical Abstraction

Some of the problems in prototyping have to do with modifying a program to keep track of the history of a data structure or to answer questions about what things happened at what time. I believe there is a new type of abstraction, called ***temporal*** or ***historical abstraction***, that can make such changes simpler. This basic idea originated with John McCarthy.

Consider a program that is simulating marriage and employment. There might be a class whose instances are people. These instances might have some structure (a data abstraction) and a protocol. One slot might record gender another marital status. Initializing the instance records the gender, and the operations of marriage, divorce, and death update marital status.

These two slots represent incommensurable things: Gender is a characteristic of an individual, and marital status slot is a characteristic of the history of an individual. The two belong in different structures: the gender in the object that represents the individual and the marital status in the object that represents the history of the individual.

The representation of the history of objects in a simulation is the history of the simulation, which happens to support a tiny protocol—all we can ask about is the present. One wonders how many data abstractions are designed to represent history or time.

The idea is to be able to refer to the computational past. Here is some code suggested by some remarks of Vladimir Lifschitz:

```
1 (defun f (n)
2   (let ((x:0 nil))
3     (dotimes (i:0 n) (setq x:0 (read))))
4   (dotimes (i n)
5     (print x:0[when (= i:0 (- n i))])))
```

There are two named variables, `x:0` and `i:0`. They are qualified names because later code will refer to them. This code is read as follows: Create a variable `x` (indexed by 0). Run a loop for `i` (indexed by 0) from 0 to $n-1$, assigning `x` to the value of READ. Run another loop from 0 to $n-1$. In this loop print the value of `x:0` when the value of `i:0` was `(- n i)`. That is, print the values that were read, backwards. No intermediate data structure needs to be created by the programmer.

Even if the need to define a data abstraction to store historical information were removed, the problem of temporal abstractions would not be solved if the programmer still was required to insert protocol invocations at the points at which the history needed to be updated. This would defeat locality. Furthermore, it would reveal implementation.

Because a prototyper wishes to modify a program as little as possible, a temporal or historical abstraction mechanism would be welcome if it were possible to isolate uses of historical abstraction from the original code.

Class-based description mechanisms are a step in this direction. For example, a triggered function defines a temporal or historical abstraction, though in a primitive way: We can define a pattern of function invocation and argument spectrum such that instances of the pattern cause record-keeping actions. It would be better to specify a language of broader patterns, possibly in terms of the sorts of inquiries that will be made about the past (or future!).

Here is a simple example of a historical abstraction; the next section has more sophisticated example:

```
(defclass person () ((gender :accessor gender) ...) ...)

(defmethod marital-status ((p person))
  (cond ((match (history p marriage-domain)
                '(* marriage divorce))
         'divorced)
        ((or (not (match (history p marriage-domain)
                         '(* marriage *)))
             (match (history p marriage-domain)
                    '(* marriage divorce)))
         'single)
        ((match (history p marriage-domain)
                '(* marriage spousal-death))
         'widowed)
        (t 'married)))
```

A pattern matching language, suggested by McCarthy, is probably simpler to work with than logic, but the language must be more expressive than I have indicated here. The form `(history p marriage-domain)` indicates we are matching events involved in the marriage domain only. The descriptor * matches any number of events of the right type. Note that the above patterns are sufficient because matching happens only on marriage-related events.

10. Idea 10: Correspondences

A good use of historical abstractions is the ***correspondence***. We've seen the use of the samefringe problem as a means to study prototyping issues. The key problem in this program is to determine whether there is a 1-1 correspondence between the leaves of two trees such that corresponding leaves are the same.

A correspondence is a mapping between two processes. Each process defines a set of objects; we use variable naming to define those things that correspond. The correspondence takes an element in the one set to the corresponding element of the second set. A correspondence also has a failure function, which is invoked if the correspondence is not 1-1. This function is invoked whenever the failure is detected.

In the following code, MAP-CORRESPONDENCE takes a function and a correspondence $c : D \rightarrow R$. It enumerates the elements of D, passing them to the function:

```
(defun samefringe (t1 t2)
  (with-names (node)
    (labels ((traverse (x)
               (cond ((atom x) x[named node])
                     (t (traverse (left x))
                        (traverse (right x))))))
      (class-let ((p1 (process) ())
                  (p2 (process) ()))
        (spawn p1 (traverse t1))
        (spawn p2 (traverse t2))
        (let ((c (correspondence {node:p1 → node:p2}
                                 (lambda ()
                                   (return-from samefringe nil)))))
          (map-correspondence
            (lambda (node)
              (unless (eq node (c node))
                (return-from samefringe nil)))
            c)
          t))))
```

11. Conclusions

It seems plain that these ideas as I've presented them add up to a complex system. However, there seems to be some grain of simplicity or at least uniformity of philosophy behind them. I think an interesting language, possibly with a non-Lisp syntax, could be built on the foundations indicated here.